February 2017

Action Language for Foundational UML (Alf)

Concrete Syntax for a UML Action Language

Version 1.1 Beta

OMG Document Number: ptc/2017-02-08

Standard document URL: http://www.omg.org/spec/ALF/1.1

Machine Readable File(s):

http://www.omg.org/spec/ALF/20170201/Alf-Syntax.xmi
http://www.omg.org/spec/ALF/20170201/Alf-Library.xmi
http://www.omg.org/spec/ALF/20120827/ActionLanguage-Profile.xmi

---

Copyright © 2010-2013 88solutions Corporation
Copyright © 2013 Commissariat a l Energie Atomique-CEA
Copyright © 2010-2017 Data Access Technologies, Inc. (Model Driven Solutions)
Copyright © 2010-2013 International Business Machines
Copyright © 2010-2013 Mentor Graphics Corporation
Copyright © 2010-2013 No Magic, Inc.
Copyright © 2010-2013 Visumpoint
Copyright © 2010-2017 Object Management Group

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® XMI® and IMM® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ , OMG Interface Definition Language (IDL)™ , and OMG SysML™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue).

Contents

Preface....xi

1 Scope....1

2 Conformance....3
- 2.1 Overview....3
- 2.2 Syntactic Conformance....3
- 2.3 Semantic Conformance....3
- 2.4 Additional Conformance Points....4

3 Normative References....5

4 Terms and Definitions....7

5 Symbols....9

6 Overview....11
- 6.1 General....11
- 6.2 Integration with UML Models....11
- 6.3 Templates....12
- 6.4 Lexical Structure....14
- 6.5 Concrete Syntax....15
- 6.6 Abstract Syntax....17
- 6.7 Mapping to Foundational UML....18
- 6.8 Organization of the Specification....18
- 6.9 Acknowledgments....19

7 Lexical Structure....21
- 7.1 General....21
- 7.2 Line Terminators....21
- 7.3 Input Elements and Tokens....21
- 7.4 White Space....22
- 7.5 Comments....22
  - 7.5.1 General....22
  - 7.5.2 Lexical Comments....22
  - 7.5.3 Documentation Comments....23
- 7.6 Names....23
- 7.7 Reserved Words....25
- 7.8 Primitive Literals....26
  - 7.8.1 General....26
  - 7.8.2 Boolean Literals....26
  - 7.8.3 Natural Literals....26
  - 7.8.4 Unbounded Value Literals....28
  - 7.8.5 String Literals....28
  - 7.8.6 Real Literals....28
- 7.9 Punctuation....29
- 7.10 Operators....30

8 Expressions....32
- 8.1 Overview....32

8.2 Qualified Names....34
8.3 Primary Expressions....39
8.3.1 Overview....39
8.3.2 Literal Expressions....40
8.3.3 Name Expressions....41
8.3.4 this Expressions....43
8.3.5 Parenthesized Expressions....44
8.3.6 Property Access Expressions....44
8.3.7 Invocation Expressions....46
8.3.8 Tuples....47
8.3.9 Behavior Invocation Expressions....50
8.3.10 Feature Invocation Expressions....54
8.3.11 Super Invocation Expressions....57
8.3.12 Instance Creation Expressions....58
8.3.13 Link Operation Expressions....62
8.3.14 Class Extent Expressions....64
8.3.15 Sequence Construction Expressions....65
8.3.16 Sequence Access Expressions....69
8.3.17 Sequence Operation Expressions....70
8.3.18 Sequence Reduction Expressions....73
8.3.19 Sequence Expansion Expressions....75
8.3.20 select and reject Expressions....77
8.3.21 collect and iterate Expressions....78
8.3.22 forAll, exists and one Expressions....79
8.3.23 isUnique Expression....81
8.4 Increment and Decrement Expressions....81
8.5 Unary Expressions....83
8.5.1 Overview....83
8.5.2 Boolean Unary Expressions....83
8.5.3 BitString Unary Expressions....84
8.5.4 Numeric Unary Expressions....85
8.5.5 Cast Expressions....86
8.5.6 Isolation Expressions....89
8.6 Binary Expressions....89
8.6.1 Overview....89
8.6.2 Arithmetic Expressions....90
8.6.3 Shift Expressions....92
8.6.4 Relational Expressions....93
8.6.5 Classification Expressions....95
8.6.6 Equality Expressions....96
8.6.7 Logical Expressions....97
8.6.8 Conditional Logical Expressions....99
8.6.9 Null-Coalescing Expressions....101
8.7 Conditional-Test Expressions....102
8.8 Assignment Expressions....105

9 Statements....111
9.1 Overview....111

9.2 Annotated Statements....114
9.3 In-line Statements....117
9.4 Block Statements....118
9.5 Empty Statements....121
9.6 Local Name Declaration Statements....122
9.7 Expression Statements....125
9.8 if Statements....126
9.9 switch Statements....129
9.10 while Statements....132
9.11 do Statements....133
9.12 for Statements....135
9.13 break Statements....139
9.14 return Statements....139
9.15 accept Statements....140
9.16 classify Statements....143

10 Units....146
10.1 Overview....146
10.2 Namespaces....153
10.3 Packages....154
10.4 Classifiers....156
10.4.1 Overview....156
10.4.2 Classes....159
10.4.3 Active Classes....162
10.4.4 Data Types....165
10.4.5 Associations....167
10.4.6 Enumerations....168
10.4.7 Signals....170
10.4.8 Activities....171
10.5 Features....175
10.5.1 Overview....175
10.5.2 Properties....175
10.5.3 Operations....179
10.5.3.1 Constructors....182
10.5.3.2 Destructors....185
10.5.4 Receptions....187

11 Standard Model Library....190
11.1 Overview....190
11.2 ActionLanguage Profile....190
11.3 Primitive Types....191
11.3.1 Natural Type....192
11.3.2 Bit String Type....192
11.4 Primitive Behaviors....192
11.4.1 Boolean Functions....193
11.4.2 Integer Functions....194
11.4.3 Real Functions....195
11.4.4 String Functions....196

11.4.5 UnlimitedNatural Functions....197
11.4.6 Bit String Functions....198
11.4.7 Sequence Functions....200
11.5 Basic Input and Output....205
11.6 Collection Functions....205
11.7 Collection Classes....208
11.7.1 Bag<T>....211
11.7.2 Collection<T>....211
11.7.3 Deque<T>....214
11.7.4 Entry....215
11.7.5 List<T>....216
11.7.6 Map<Key, Value>....219
11.7.7 OrderedSet<T>....222
11.7.8 Queue<T>....225
11.7.9 Set<T>....228

12 Common Abstract Syntax....230
12.1 Overview....230
12.2 Class Descriptions....232
12.2.1 AssignedSource....232
12.2.2 DocumentedElement....233
12.2.3 ElementReference....234
12.2.4 ExternalElementReference....234
12.2.5 InternalElementReference....234
12.2.6 SyntaxElement....235

13 Expressions Abstract Syntax....237
13.1 Overview....237
13.2 Class Descriptions....244
13.2.1 ArithmeticExpression....244
13.2.2 AssignmentExpression....246
13.2.3 BehaviorInvocationExpression....251
13.2.4 BinaryExpression....252
13.2.5 BitStringUnaryExpression....253
13.2.6 BooleanLiteralExpression....254
13.2.7 BooleanUnaryExpression....254
13.2.8 CastExpression....255
13.2.9 ClassExtentExpression....256
13.2.10 ClassificationExpression....257
13.2.11 CollectOrIterateExpression....258
13.2.12 ConditionalLogicalExpression....259
13.2.13 ConditionalTestExpression....260
13.2.14 EqualityExpression....261
13.2.15 Expression....263
13.2.16 ExtentOrExpression....264
13.2.17 FeatureInvocationExpression....265
13.2.18 FeatureLeftHandSide....266
13.2.19 FeatureReference....267

13.2.20 ForAllOrExistsOrOneExpression....268
13.2.21 IncrementOrDecrementExpression....268
13.2.22 InstanceCreationExpression....270
13.2.23 InvocationExpression....272
13.2.24 IsolationExpression....274
13.2.25 IsUniqueExpression....275
13.2.26 LeftHandSide....276
13.2.27 LinkOperationExpression....276
13.2.28 LiteralExpression....277
13.2.29 LogicalExpression....278
13.2.30 NameBinding....279
13.2.31 NamedExpression....280
13.2.32 NamedTemplateBinding....281
13.2.33 NamedTuple....281
13.2.34 NameExpression....282
13.2.35 NameLeftHandSide....283
13.2.36 NaturalLiteralExpression....285
13.2.37 NullCoalescingExpression....286
13.2.38 NumericUnaryExpression....287
13.2.39 OutputNamedExpression....288
13.2.40 PositionalTemplateBinding....288
13.2.41 PositionalTuple....289
13.2.42 PropertyAccessExpression....289
13.2.43 QualifiedName....290
13.2.44 RealLiteralExpression....292
13.2.45 RelationalExpression....293
13.2.46 SelectOrRejectExpression....294
13.2.47 SequenceAccessExpression....295
13.2.48 SequenceConstructionExpression....296
13.2.49 SequenceElements....297
13.2.50 SequenceExpansionExpression....298
13.2.51 SequenceExpressionList....299
13.2.52 SequenceOperationExpression....300
13.2.53 SequenceRange....302
13.2.54 SequenceReductionExpression....303
13.2.55 ShiftExpression....304
13.2.56 StringLiteralExpression....305
13.2.57 SuperInvocationExpression....305
13.2.58 TemplateBinding....306
13.2.59 TemplateParameterSubstitution....306
13.2.60 ThisExpression....307
13.2.61 Tuple....308
13.2.62 UnaryExpression....309
13.2.63 UnboundedLiteralExpression....309

14 Statements Abstract Syntax....312
14.1 Overview....312
14.2 Class Descriptions....316

14.2.1 AcceptBlock....316
14.2.2 AcceptStatement....317
14.2.3 Annotation....318
14.2.4 Block....319
14.2.5 BlockStatement....320
14.2.6 BreakStatement....321
14.2.7 ClassifyStatement....321
14.2.8 ConcurrentClauses....322
14.2.9 DoStatement....323
14.2.10 EmptyStatement....324
14.2.11 ExpressionStatement....325
14.2.12 ForStatement....325
14.2.13 IfStatement....327
14.2.14 InLineStatement....328
14.2.15 LocalNameDeclarationStatement....329
14.2.16 LoopVariableDefinition....330
14.2.17 NonFinalClause....332
14.2.18 QualifiedNameList....333
14.2.19 ReturnStatement....334
14.2.20 Statement....334
14.2.21 SwitchClause....336
14.2.22 SwitchStatement....337
14.2.23 WhileStatement....338

15 Units Abstract Syntax....340
15.1 Overview....340
15.2 Class Descriptions....342
15.2.1 ActiveClassDefinition....342
15.2.2 ActivityDefinition....343
15.2.3 AssociationDefinition....344
15.2.4 ClassDefinition....345
15.2.5 ClassifierDefinition....345
15.2.6 ClassifierTemplateParameter....346
15.2.7 DataTypeDefinition....347
15.2.8 ElementImportReference....347
15.2.9 EnumerationDefinition....348
15.2.10 EnumerationLiteralName....349
15.2.11 FormalParameter....349
15.2.12 ImportedMember....350
15.2.13 ImportReference....350
15.2.14 Member....351
15.2.15 NamespaceDefinition....353
15.2.16 OperationDefinition....354
15.2.17 PackageDefinition....356
15.2.18 PackageImportReference....357
15.2.19 PropertyDefinition....357
15.2.20 ReceptionDefinition....358
15.2.21 SignalDefinition....359

15.2.22 SignalReceptionDefinition....360
15.2.23 StereotypeAnnotation....360
15.2.24 TaggedValue....361
15.2.25 TaggedValueList....362
15.2.26 TypedElementDefinition....362
15.2.27 UnitDefinition....364

16 Common Mapping....366
16.1 General....366
16.2 Syntax Elements....366
16.3 Documented Elements....366
16.4 Element References....366
16.5 Assigned Sources....366

17 Expressions Mapping....368
17.1 General....368
17.2 Qualified Names....368
17.3 Literal Expressions....368
17.4 Name Expressions....368
17.5 this Expressions....369
17.6 Property Access Expressions....369
17.7 Invocation Expressions....369
17.8 Tuples....370
17.9 Behavior Invocation Expressions....370
17.10 Feature Invocation Expressions....371
17.11 Super Invocation Expressions....372
17.12 Instance Creation Expressions....372
17.13 Link Operation Expressions....373
17.14 Class Extent Expressions....374
17.15 Sequence Construction Expression....374
17.16 Sequence Access Expressions....375
17.17 Sequence Operation Expressions....375
17.18 Sequence Reduction Expression....375
17.19 Sequence Expansion Expressions....376
17.20 Increment and Decrement Expressions....377
17.21 Unary Expressions....377
17.22 Binary Expression....379
17.23 Conditional-Test Expressions....382
17.24 Assignment Expressions....383

18 Statements Mapping....387
18.1 General....387
18.2 In-Line Statements....387
18.3 Block Statements....387
18.4 Empty Statements....388
18.5 Local Name Definition Statements....388
18.6 Expression Statements....388
18.7 if Statements....388
18.8 switch Statements....389

18.9 while Statements....389
18.10 do Statements....390
18.11 for Statements....391
18.12 break Statements....392
18.13 return Statements....392
18.14 accept Statements....392
18.15 classify Statements....393

19 Units Mapping....395
19.1 General....395
19.2 Namespace Definitions....395
19.3 Package Definitions....395
19.4 Classifier Definitions....396
19.5 Class Definitions....396
19.6 Active Class Definitions....397
19.7 Data Type Definitions....397
19.8 Association Definitions....397
19.9 Enumeration Definitions....397
19.10 Signal (and Signal Reception) Definitions....397
19.11 Activity Definitions....398
19.12 Typed Element Definitions....398
19.13 Formal Parameters....399
19.14 Property Definitions....399
19.15 Operation Definitions....399
19.16 Reception Definitions....400

Annex A: Semantic Integration with State Machines and Composite Structure....401
A.1 Overview....401
A.2 State Machines....401
A.3 Composite Structure....405

Annex B: Extended Examples....409
B.1 Quicksort Activity....409
B.1.1 Quicksort Functional Implementation....409
B.1.2 Quicksort “In Place” Implementation....413
B.2 Online Bookstore....415
B.2.1 Graphical Model for Ordering....415
B.2.2 Alf Representation of Entry Behaviors....417
B.2.2.1 Activity EstablishCustomer....418
B.2.2.2 Activity ProcessCharge....419
B.2.2.3 Activity DeclineCharge....420
B.2.2.4 Activity PackAndShip....420
B.2.2.5 Activity NotifyOfDelivery....420
B.2.3 Alf Representation of the Ordering Model....420
B.2.3.1 Package Ordering....421
B.2.4 Class Order....422
B.3 Property Management Service....425
B.3.1 The Property Management Model....425
B.3.2 Data Model....426

B.3.3 Message Model....428
B.3.3.1 Request Messages....428
B.3.3.2 Reply Messages....429
B.3.4 Service Model....430
B.3.5 Property Management Service Methods....431
B.3.5.1 Property....431
B.3.5.2 Identifier Factory....432
B.3.5.3 Property Management Service Implementation....433
B.4 Alf Standard Library Collection Classes Implementation....440
B.4.1 Introduction....440
B.4.2 CollectionClasses::Impl....440
B.4.3 CollectionClasses::Impl::CollectionImpl....441
B.4.4 CollectionClasses::Impl::OrderedCollectionImpl....443
B.4.5 CollectionClasses::Impl::Set....444
B.4.6 CollectionClasses::Impl::OrderedSet....446
B.4.7 CollectionClasses::Impl::Bag....448
B.4.8 CollectionClasses::Impl::List....450
B.4.9 CollectionClasses::Impl::Queue....453
B.4.10 CollectionClasses::Impl::Deque....455
B.4.11 CollectionClasses::Impl::Map....457

Annex C: Consolidated LL Grammar....460
C.1 Introduction....460
C.2 Lexical Analyzer....460
C.3 Parser....464

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. Specifications are available from this URL:

http://www.omg.org/spec

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

CORBA/IIOP

Data Distribution Services

Specialized CORBA

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

UML, MOF, CWM, XMI

UML Profile

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

CORBAServices

CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Ave
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

1 Scope

The Action Language for Foundational UML (or "Alf") is a textual surface representation for UML modeling elements. The execution semantics for Alf are given by *mapping* the Alf concrete syntax to the abstract syntax of the standard Foundational Subset for Executable UML Models (known as "Foundational UML" or "fUML"). The result of executing an Alf input text is thus given by the semantics of the fUML model to which it is mapped, as defined in the fUML specification.

A primary goal of an action language is to act as the surface notation for specifying executable behaviors within a wider model that is primarily represented using the usual graphical notations of UML. For example, this might include methods on the operations of classes or transition effect behaviors on state machines.

However, Alf also provides an extended notation that may be used to represent structural modeling elements. Therefore, it is possible to represent a UML model entirely using Alf, though Alf syntax only directly covers the limited subset of UML structural modeling available in the fUML subset.

Key guiding principles in the design of Alf include the following:

- Alf has a largely C-legacy ("Java like") syntax, since that is most familiar to the community that programs detailed behaviors. Nevertheless, Alf allows UML textual syntax when it exists (e.g., colon syntax for typing, double colon syntax for name qualification, etc.).
- Alf does not require graphical models to change in order to accommodate use of the action language (e.g., special characters are allowed in names, arbitrary names are allowed for constructors, etc.). Further, while Alf maps to the fUML subset in order to provide its execution semantics, it is usable in context of models not limited to the fUML subset.
- Alf provides a naming system that is based on UML namespaces for referencing elements outside of an activity but also provides for the consistent use of local names to reference flows of values within an activity.
- Alf uses an implicit type system that allows but does not require the explicit declaration of typing within an activity, always providing for static type checking, based at least on typing declared in the structural model elements.
- Alf has the expressivity of OCL in the use and manipulation of sequences of values. These sequence expressions are fully executable in terms of fUML expansion regions, allowing the simple and natural specification of highly concurrent computations.
- While the primary goal of Alf is to be an action language, Alf also provides concrete syntax for structural modeling, largely within the bounds of the fUML subset.

2 Conformance

2.1 Overview

There are two main aspects of conformance to the Alf standard:

- *Syntactic Conformance.* Alf input text must conform syntactically to one of the levels defined below in 2.2.
- *Semantic Conformance.* A conforming modeling tool must process syntactically conforming Alf text in one of the ways defined below in 2.3.

In addition, 2.4 defines two further mandatory conformance points.

2.2 Syntactic Conformance

Clause 6 discusses the overall requirements for processing of Alf input text. For the purposes of the present discussion of syntactic conformance, "syntactic processing" includes lexical analysis, as specified in Clause 7, the parsing of the various Alf features specified in Clauses 8 through 10 and the static semantic analysis of Alf abstract syntax trees specified in Clauses 12 through 15.

There are three levels of syntactic conformance, depending on whether syntactic processing must be supported for all features specified in Clauses 8 through 10, or only some subset of them.

1. *Minimum Conformance.* Conformance at this level requires the ability to process a subset of the syntax defined in Clauses 8 and 9, but none of the syntax defined in Clause 10. The exact subset that must be supported is identified in each of the syntactic grammar specifications in Clauses 8 and 9. The intent of Minimum Conformance is to provide a subset of Alf that is usable for writing textual action language snippets within a larger graphical UML model and that includes only the capabilities available in a traditional, procedural programming language.
2. *Full Conformance.* Conformance at this level requires the ability to process all the syntax defined in Clauses 8 and 9, but none of the syntax defined in Clause 10. At Full Conformance, Alf provides a complete action language for representing behavior within a structural model represented in a typical UML modeling environment outside of Alf (see 6.2).
3. *Extended Conformance.* Conformance level requires the ability to process all the syntax defined in Clauses 8, 9 and 10. This includes not only the action language capabilities of Full Conformance, but also the structural modeling capabilities defined in Clause 10.

2.3 Semantic Conformance

The execution semantics for Alf are described informally in Clauses 8 through 10 and specified by a formal mapping to fUML in Clauses 16 through 19. A conforming execution tool must implement the specified semantics for the syntactic subset of Alf to which the tool conforms.

The execution semantics for all Alf constructs are formally specified via their mapping to fUML (see 6.7) and the execution of the resulting fUML model per the semantics of the fUML Specification. However, a conforming modeling tool does not need to actually perform this mapping in order to execute Alf input text. Indeed, there are three ways in which the tool may implement the specified Alf execution semantics.

1. *Interpretive Execution.* The modeling tool directly interprets and executes the Alf input text.
2. *Compilative Execution.* The modeling tool compiles the Alf text to a UML model conforming to the fUML subset (at level L3) and executes that per the semantics specified in the fUML Specification, perhaps in the context of the execution of a larger model that may not conform to the fUML subset.

   NOTE. The compiled model resulting from the Alf text does *not* have to be the same as that resulting from the standard mapping to fUML defined in the Alf specification (though it must have an equivalent effect—see below), but it must be conformant to the fUML subset and thus executable by a fUML-conforming execution tool.
3. *Translational Execution.* The modeling tool translates the Alf text, as well as any surrounding larger UML model as appropriate, into some executable form on a non-UML target platform, and executes the translation on that platform.

In all cases, the portion of the execution corresponding to an Alf input text must have the *equivalent effect* to mapping that text to fUML per the Alf specification and executing the resulting model per the semantics specified in the fUML Specification. For the purposes of semantic conformance as defined here, this means that:

- The effect of executing the Alf text within any larger containing model is per the equivalent fUML semantics.
- Any visible effect produced by executing the Alf text within the target execution environment is per the equivalent fUML semantics (where, for the purposes of fUML semantics, the *execution environment* is as defined in Clause 2 of the fUML Specification).

2.4 Additional Conformance Points

There are two additional mandatory conformance points for this standard:

1. *Template Semantics.* Every conformant modeling tool must conform to the template semantics specified in 6.3.
2. *Library Implementation.* Every conformant modeling tool must provide an implementation of the Alf Standard Modeling Library, conforming to the specification given in Clause 11.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification.

- *OMG Unified Modeling Language (OMG UML), Infrastructure,* Version 2.4.1 (http://www.omg.org/spec/UML/2.4.1/Infrastructure) – Referenced in the following as "UML Infrastructure"
- *OMG Unified Modeling Language (OMG UML) Superstructure,* Version 2.4.1 (http://www.omg.org/spec/UML/2.4.1/Superstructure) – Referenced in the following as "UML Superstructure"
- *Semantics of a Foundational Subset for Executable UML Models (fUML),* Version 1.3 (http://www.omg.org/spec/FUML/1.3) – Referenced in the following as "fUML Specification"
- *Object Constraint Language,* Version 2.2 (http://www.omg.org/spec/OCL/2.2) – Referenced in the following as "OCL Specification"

4 Terms and Definitions

Execution Semantics

For the purposes of this specification, the behavioral semantics of UML constructs that specify operational action over time, describing or constraining allowable behavior in the domain being modeled. (From the fUML Specification.)

Execution Tool

Any tool that is capable of executing any valid UML model that is based on the foundational subset and expressed as an instantiation of the UML 2.0 abstract syntax metamodel. This may involve direct interpretation of UML models and/or generation of equivalent computer programs from the models through some kind of automated transformations. Such a tool may also itself be concurrent and distributed. (From the fUML Specification.)

Foundational Subset

The subset of UML to which execution semantics are given in order to provide a foundation for ultimately defining the execution semantics of the rest of UML. (From the fUML Specification.)

Modeling Environment

A user environment provided by a modeling tool that allows for the creation and modification of a UML model. In a modeling tool that is an execution tool, the modeling environment may also allow for direct execution of the model.

Modeling Tool

Any tool that allows for the creation and management of UML models. A modeling tool that allows those models to be executed is also an execution tool.

5 Symbols

There are no symbols or abbreviated terms necessary for the understanding of this specification.

6 Overview

6.1 General

An Alf *input text* is a concrete representation for UML model elements in the Foundational UML (fUML) abstract syntax subset (see fUML Specification, Clause 7). Such an input text may be part of a wider UML model only parts of which are represented in Alf, or it may be the representation of an entire model in its own right. In either case, this specification defines how concrete Alf input text is processed into an abstract syntax representation of UML model elements.

6.2 Integration with UML Models

The UML Superstructure specification defines the standard graphical and textual notations used to express a UML model. In this context, Alf can be used as an alternative textual notation to represent portions of the overall model. The following are the ways in which Alf may be so used:

- As described in 8.1, an Alf *expression* may be used any place a UML *value specification* is allowed. This may be done either by including the Alf text as the body of a UML *opaque expression* or the Alf text may be compiled into an equivalent UML activity to act as the specification of such an expression. In addition, there are special syntactic forms for instance creation and sequence construction expressions that do not require the explicit annotation of redundant type information in Alf expressions that are used to specify the default values for properties or parameters.
- As described in 9.1, a sequence of Alf *statements* may be used in two ways:
  1. To define the behavior of a UML *action* within an activity or interaction model. The Alf text may be included as the body of a UML *opaque action* or it may be compiled into an equivalent UML *structured activity node* (which is a kind of action).
  2. To define the behavior of a complete UML *behavior*. The Alf text may be included as the body of a UML *opaque behavior* or it may be compiled into an equivalent UML activity (which is a kind of behavior).
- As described in 10.1, an Alf *model unit* may be used to represent the model of a classifier or package that is intended to be individually referenced as a named element. Such a model unit may represent an entire UML model (at least within the limits of the fUML subset) or it may represent a model element (such as a class or standalone activity) intended to be used within some larger model.

Since an Alf text can be processed into a UML abstract syntax representation, a portion of a model represented in Alf can always be integrated into a larger model on that basis (as noted above), regardless of the surface representation of any portion of the model. Nevertheless, even when such compilation is done, it will generally still be desirable to also store the original Alf text in the model, since it would not otherwise be possible to exactly reproduce that text (with user formatting, etc.) from the model representation. This is done by attaching a comment to the top-level element resulting from the processing of an Alf model unit with the Alf text for the unit as the body of the comment and the applied stereotype «`TextualRepresentation`» with tagged value `{language = "Alf"}`. This stereotype is from the standard `ActionLanguage` profile defined in 11.2.

6.3 Templates

Templates, defined in 17.5 of the UML Superstructure, are not included in the fUML subset. Nevertheless, Alf text may be used as snippets within the context of a wider model that is a template. Since template parameters are tied to parameterable elements that are used as regular model elements within the context of the template model (see UML Superstructure, 17.5.1), Alf text may refer to these elements in the usual way.

However, templates are particularly useful for modeling parameterized types, and they are, in fact, used in this way in the `CollectionClasses` model in the Alf Standard Model Library (see 11.7). But Alf text that uses types that are instantiations of such templates still needs to be mapped to fUML in order to provide its formal semantics. This requires a way to define the semantics of template binding in terms of constructs available in the fUML subset.

According to the UML Superstructure specification (17.5.2 TemplateableElement, under Semantics), "The semantics of a [template] binding relationship is equivalent to the model elements that would result from copying the contents of the template into the bound element, replacing any elements exposed as a template parameter with the corresponding element(s) specified as actual parameters in this binding." This section also states that "In a canonical model a bound element does not explicitly contain the model elements implied by expanding the templates it binds to, since those expansions are regarded as derived." However, by actually carrying out the expansion, one can obtain an equivalent model that does not explicitly refer to the original template or the binding of its parameters. (This is analogous to the way other forms of derivation in the UML abstract syntax model are handled in fUML—see fUML Specification, 8.1.)

In order to make the explicit *copy semantics* of template binding more precise, an *equivalent bound element* may be constructed for any element with a template binding by the following steps:

1. Copy the template associated with the template signature that is the target of the template binding. For the present purposes, a *copy* of a model element is an instance of the same metaclass as the original model element that has the same values as the original element for all non-composite properties (owned attributes and association ends) and copies (in the same sense) of the values from the original element for all composite properties.

2. If the copy specializes any elements that are templates, then redirect the generalization relationships to equivalent bound elements for the general elements, using the same template binding. If the copy is an operation that has an associated method that is also a template, then replace that method with an equivalent bound element using the same template binding.

   NOTE. The UML Superstructure does not address the issue of methods of template operations. However, it is necessary for the method of a template operation to also be a template, presumably with the same template parameters as the operation. In particular, operation template parameters are typically used to parameterize the types of operation parameters, but the method of an operation does not directly reference the parameters of the operation that specifies it. Rather, the method has its own parameter list, which must match that of the operation (see UML Superstructure, 13.3.2). The types of the method parameters would thus need to be separately templated to match the template parameterization of the operation.

3. For each element owned directly or indirectly by the copy, replace any reference to the parametered element of a template parameter of the copy with a reference to the actual element associated with

the parameter in the template binding (if any). If an actual element has a template binding itself, then reference the equivalent bound element.

4. Remove all template parameters that are referenced in the template binding from the template signature of the copy. If this would remove all template parameters from the template signature, then remove the template signature entirely.

The syntax for template binding in Alf is defined in 8.2. Only a limited set of template bindings may be so notated in Alf. Specifically:

- The binding must specify an actual element for every template parameter of the template.
- The element being bound must not have template parameters of its own.

Thus, the equivalent bound element (as defined above) for a template binding notated in Alf will always be a directly usable non-template element. However, in this context, it is important that two identical template bindings be considered to result in the *same* element. Otherwise, every template binding would lead to a separate instantiation of the template, even if the bindings were equivalent, which would have undesirable consequences.

For example, a set of integers may be notated in Alf as `Set<Integer>`, using the standard template collection class `Set` and the standard primitive type `Integer`. If each occurrence in a model of the Alf text "`Set<Integer>`" resulted in a different equivalent bound element, then an object created using one such occurrence would not be type compatible with, say, a formal parameter of an operation whose type is given by another such occurrence. Clearly this is not desirable.

Therefore, the template bindings within a model must be replaced as a whole using the following steps:

1. Partition the set of all elements with template bindings in the model into disjoint subsets of elements with *identical* bindings. Two template bindings are considered identical if they have the same set of parameter substitutions. Two parameter substitutions are considered to be the same if they reference the same formal parameter and actual element.

   NOTE. The sameness of parameter substitutions is determined by the elements referenced, regardless of the names that may be used to reference those elements in the Alf text for a template binding. That is, the use of unqualified names, qualified names or aliases is irrelevant to the determination of whether two template bindings are identical, so long as corresponding names resolve to the same element.

2. For each subset, create a *single* equivalent bound element (as defined above), starting with any one member of the subset.

3. Replace any reference to any element in the model with a template binding with a reference to the equivalent bound element for its subset.

In order to simplify the identification of equivalent bound elements after the above substitutions are carried out, Alf defines a standard naming convention for such elements, constructed as follows:

1. Take the fully qualified name of each actual element in the template binding and replace all "`::`" separators with "`$`" characters. If the actual element is itself a template binding, then use the name of the equivalent bound element. If the actual element is empty (null), then use "any" as the name of the actual element. (While the Alf syntax for template bindings given in 8.2 does not actually allow any to be used as a template argument, such an argument can result from the rules for the implicit binding for a template behavior as given in 8.3.9)

2. Concatenate the modified names, separated by "_" (one underscore), and prepended and postpended with "__" (two underscores).
3. Concatenate "`$$`", the name of the target template of the template binding and the argument name list from 2 to produce the standard name of the equivalent bound element.

For example, the standard name for the equivalent bound element for `Set<Integer>` is

```
$$Set__UML$AuxiliaryConstructs$PrimitiveTypes$Integer__
```

Note that the qualified name for an element is determined by its *owning* namespace. Therefore, even though the name "`Integer`" resolves in Alf to "`Alf::Library::PrimitiveTypes:: Integer`", this element is just an import of "`UML::AuxiliaryTypes::PrimitiveTypes:: Integer`" (see 11.3), and it is the latter qualified name that is used.

Since the initial copy of the template model element also copies the reference to the namespace of the original template, the equivalent bound element is considered to be added to that namespace. A modeling environment must disallow any user-created element in a namespace with template elements with a name that would conflict with a standard equivalent bound element name created as defined above.

NOTE. The concept of equivalent bound elements defined above is intended to provide a specification of the semantics of template instantiation compatible with execution semantics that are defined only on the fUML subset. It is not required that a conforming implementation actually physically generate equivalent bound elements in order to execute Alf text, particularly if that implementation semantically conforms through interpretive or translational execution (see 2.3). However, an implementation that conforms through compilative execution must produce a UML model conforming to the fUML subset, in which case the implementation would, in fact, need to replace templates and bindings with equivalent elements as described here (or similar elements with an equivalent effect, as discussed in 2.3).

6.4 Lexical Structure

The *lexical structure* of Alf defines how the string of characters in an Alf input text is divided into a set of *input elements.* Such input elements can be categorized as whitespace, comments, or tokens.

Lexical analysis is the process of converting an Alf input text into a corresponding stream of input elements. After lexical analysis, whitespace and comments are discarded and only tokens are retained for the subsequent step of parsing. Lexical analysis for Alf is thus essentially the same as is done for the processing of any typical textual programming language.

The Alf lexical structure is specified by a lexical grammar in which characters are terminal elements and the input elements resulting from lexical analysis are non-terminal elements. The lexical grammar is defined using an Extended Backus-Naur Form (EBNF) notation, whose conventions are given in Table 6.1.

Table 6.1 EBNF Notation Conventions

Terminal element	`"terminal"`
Non-terminal element	`NonterminalElement`
Sequential elements	`Element1 Element2`
Alternative elements	`Element1 | Element2`
Optional element (zero or one)	`[ Element ]`
Repeated element (zero or more)	`{ Element }`
Production definition	`NonterminalElement = …`

* The escape sequences given in Table 7.1 are also used to represent the corresponding special characters within terminal elements in the EBNF notation.

6.5 Concrete Syntax

The *concrete syntax* of Alf defines how lexical tokens are grouped into an *abstract syntax tree. Parsing* is the process of constructing an abstract syntax tree from the tokens produced by the lexical analysis of an Alf text. The parsing of an Alf input text is thus essentially the same as is done for the processing of any typical textual programming language.

The Alf concrete syntax is specified by a syntactic grammar whose definition is also based on the EBNF notation given in Table 6.1. However, elements of the productions in the syntactic grammar are further annotated to indicate how an abstract syntax tree is to be constructed during parsing. The EBNF specification of Alf syntax provides the basis for the context-free parsing of an Alf input text. Context-dependent constraints are then enforced on the abstract syntax representation.

As described further below in 6.6, the abstract syntax for Alf is specified as a UML class model. A production in the syntactic grammar results in the *synthesis* of an instance of a class in the abstract syntax. The production definition defines a name for the instance being synthesized, which is used in the body of the production, and declares the class of the instance.

For example, the production

```
LocalNameDeclarationStatement(s: LocalNameDeclarationStatement)
      = NameDeclaration(s) "=" InitializationExpression(s.expression) ";"
```

declares that it synthesizes an object called `s` of class `LocalNameDeclarationStatement`. Note the name of the class of the synthesized object is often the same as the non-terminal defined by the production, as in this example, but this is not always the case.

Annotations are parenthesized in the body of a production. There are several forms of annotation, as shown in Table 6.2. For example, the body of the production given above may be read as follows: a `LocalNameDeclarationStatement` consists of a `NameDeclaration` (which is parsed into the object `s`), followed by “=”, followed by an `InitializationExpression` (the abstract syntax representation of which is assigned to the `expression` attribute of object `s`), followed by a “`;`”.

Table 6.2 Abstract Syntax Synthesis Notation

Non-terminal object constraint	`NonterminalElement(x)`	The object `x` is constrained to be the same as the object synthesized for the immediately preceding non-terminal element.
Non-terminal property constraint	`NonterminalElement(x.p)`	The object synthesized for the immediately preceding non-terminal element is added to the list of values of property `p` of object `x`. If the non-terminal element is a lexical token, then the string image of the token is added to the property.
Terminal string constraint	`"terminal"(x)`	The synthesized "object" x, which must have the primitive type `String`, is constrained to have the string image of the immediately preceding terminal element as it's value.
Terminal property constraint	`"terminal"(x.p)`	The string image of the immediately preceding terminal element is added to the list of values of property `p` of object `x`.
General constraint	`(expr)`	The OCL constraint expression `expr` must be true.

If an annotation appears in an alternative or optional element group, then it only applies if the content of that group actually applies during parsing. If an annotation appears in a repeating group, then it applies during each repeated application of the content of that group.

For example, the following is the production for the non-terminal `NameDeclaration` used in the body of the production for `LocalNameDeclarationStatement` above.

```
NameDeclaration(s: LocalNameDeclarationStatement)
      = "let" Name(s.name) ":" TypeName(s.typeName)
        [ MultiplicityIndicator (s.hasMultiplicity=true) ]
      | TypeName(s.typeName)
        [ MultiplicityIndicator (s.hasMultiplicity=true) ] Name(s.name)
```

According to this production, a `NameDeclaration` may have one of two alternative forms:

- The terminal element "`let`", followed by a `Name` (a lexical token whose string image is assigned to the `name` property of object `s`), followed by a ":", followed by a `TypeName` (whose value is assigned to `s.typeName`), optionally followed by a `MultiplicityIndicator` (if there is a `MultiplicityIndicator`, then `s.hasMultiplicity` must be true).
- A `TypeName` (whose value is assigned to `s.typeName`) optionally followed by a `MultiplicityIndicator` (if there is a `MultiplicityIndicator`, then `s.hasMultiplicity` must be true), followed by a `Name` (whose string image is assigned to `s.name`).

Finally, consider the production

```
ColonQualifiedName(q: QualifiedName)
      = Name(q.name) "::" { Name(q.name) "::" } Name(q.name)
```

In this case, the first name is assigned to the property `q.name`, and then subsequent names, if any, are appended as additional values of that same property. Clearly, for this to be valid, the property `QualifiedName::name` must have a multiplicity upper bound of `*`.

Technically, the Alf concrete syntax is specified (as described above) using a simplified form of an *attributive grammar.* Each production has a single *synthesized attribute.* In an attributive grammar, the values for synthesized attributes are passed upwards in the parse tree. The Alf concrete grammar does not include any *inherited attributes,* however. Inherited attributes are passed downwards in the parse tree in order to enforce context-sensitive constraints. For Alf, such constraints are instead specified on the abstract syntax representation after parsing.

6.6 Abstract Syntax

The *abstract syntax* for Alf is a UML class model of the tree of objects synthesized from parsing an Alf text (as described above in 6.5). The Alf concrete syntax is context free and parsing based on this syntax results in a strictly hierarchical parse tree. The synthesized abstract syntax tree is an abstraction of the complete parse tree—for example, punctuation symbols are not included in the abstract syntax tree—but it is still a hierarchical tree structure.

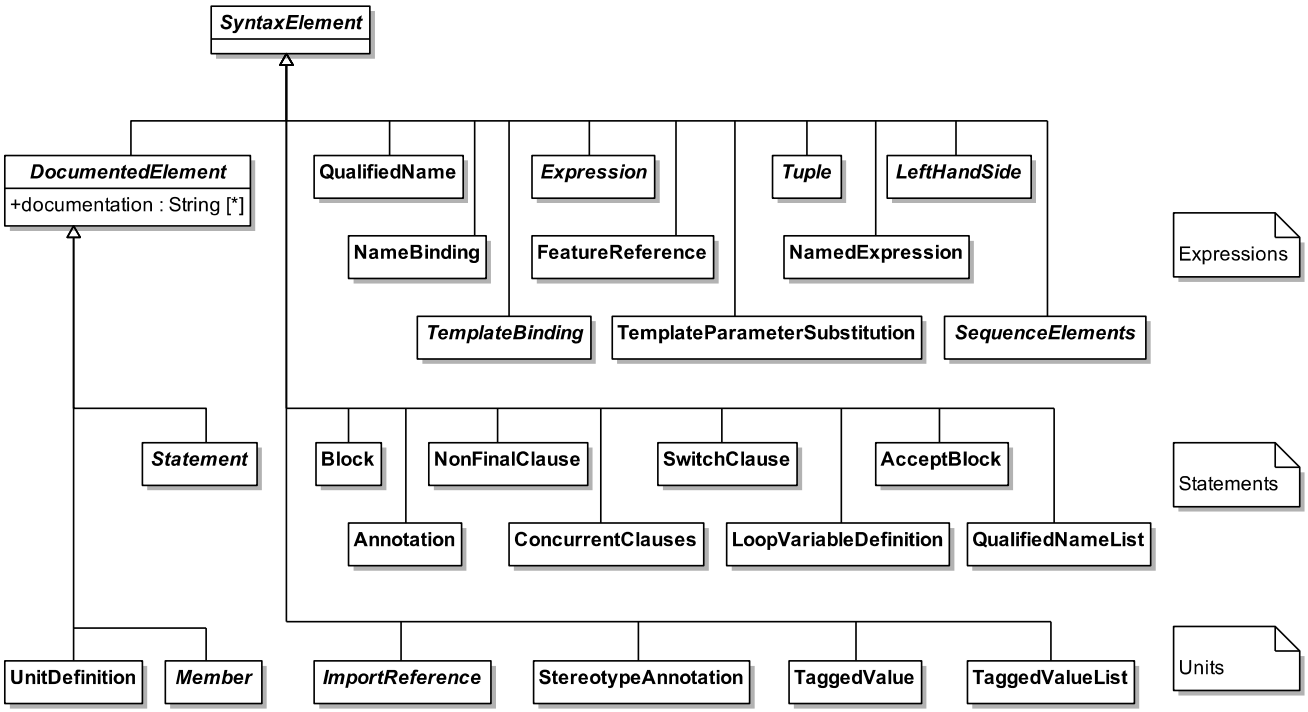

Figure 6.1 Top-Level Syntax Element Classes

The nodes of an abstract syntax tree are objects known as *syntax elements.* Every syntax element class descends from the root abstract class SyntaxElement. Figure 6.1 shows this root class and its top-level descendents. Note that certain of the classes shown in Figure 6.1 are subclasses of the intermediate DocumentedElement class. These classes represent elements that may be annotated with documentation

comments (see 7.5.3) that are preserved in the ultimate fUML model, as opposed to lexical comments (see 7.5.2), which are not so preserved.

While the abstract syntax tree synthesized from parsing an Alf text is strictly hierarchical, there are important non-hierarchical relationships and constraints between Alf elements that may be determined solely from static analysis of the abstract syntax tree. Such *static semantic analysis* is also common in the processing of a typical programming language, particularly for resolving names and type checking. However, the analysis for Alf is somewhat different than for the typical case, since it is used to gather additional information required for mapping to fUML, rather than generating machine code as in the case of a programming language.

6.7 Mapping to Foundational UML

The final step in the processing of an Alf input text is the mapping of the abstract syntax tree for the text, completed with derived property values from static semantic analysis, to a representation in terms of the fUML abstract syntax; that is, as interrelated instances of the abstract syntax metaclasses specified in Clause 7 of the fUML Specification. The fUML abstract syntax representation can be built via a generally depth-first traversal of the Alf abstract syntax tree, with the static semantic information providing non-hierarchical relationships across the tree structure.

Concurrency

Most mainstream programming languages are based on an inherently sequential model of computation. Statements are executed sequentially, argument expressions are evaluated in lexical order, etc. In many cases, however, such specifically fined-grained sequential execution (especially in expressions) is entirely inessential to the computation being carried out and unnecessarily restricts the options for how the computation can be implemented. This has become a problem, for example, in implementing such languages in a way that takes advantage of the increasing amount of parallelism available in mainstream hardware platforms.

On the other hand, the execution semantics for activities in fUML are inherently concurrent. In general, the semantics of UML activities place no restriction on the concurrent execution of nodes within the activity, other than that imposed by the explicit object and control flows between those nodes, and the Alf mapping to fUML takes advantage of this concurrency in many places, particularly in the mapping of expressions. For example, the argument expressions in an operation invocation or an arithmetic expression are evaluated concurrently, rather than sequentially.

However, as noted in fUML Specification, 8.5.2.1 (under the Threading Model heading), the concurrency in the fUML semantics “does not require the implementation of actual *parallelism* in a conforming execution tool. It simply means that such parallelism is *allowed* and that the execution semantics provide no further restriction on the serialization of execution across concurrent [computations].”

Correspondingly, any place in the description of the Alf semantics that computations are specified to be concurrent, this should be understood to simply mean that there is no requirement that they be carried out in any particular sequential order. If desired, though, a conformant tool may still carry out the computations completely sequentially, in any desired order. Or it may actually carry out some or all of the computations in parallel, either virtually (e.g., in separate threads) or physically (on separate processors).

6.8 Organization of the Specification

This Alf specification document is organized into five parts.

The first part comprises the initial clauses, through Clause 6. These clauses contain introductory material and an overview of how the Alf language is defined.

The second part contains a complete description of the Alf language. Clause 7 defines the lexical structure for Alf, which specifies how lexical analysis is to be carried out for an Alf input text. Clauses 8 through 10 provide the core description of Alf constructs in the syntactic areas of expressions, statements and units. Within these clauses, the for each construct formally defines its concrete syntax and the synthesis of the abstract syntax from that. Each also includes examples of the use of the construct and an informal description of the semantics of the construct. Clause 11 defines a standard model library that must be provided with any Alf implementation. This library includes the fUML Foundational Model Library (see Clause 9 of the fUML Specification) plus additional primitive types and primitive behaviors. It also includes a set of collection classes and the profile used in attaching Alf text to the models to which that text was mapped.

The third part provides the complete definition of the abstract syntax of Alf, including the formal specification of the static semantics for Alf in terms of additional derived properties and constraints. Each of the four clauses in this part correspond to a package in the top-level decomposition of the Alf abstract syntax model.

The fourth part gives the formal mapping of the abstract syntax, as annotated during static semantic analysis, to the fUML subset of the UML abstract syntax. The four clauses in this part also correspond to the abstract syntax packages.

The fifth part comprises the annexes, all of which are informative, not normative. Annex A discusses the integration of Alf execution semantics with the non-fUML execution semantics of state machine and composite structure models. Annex B provides a number of extended examples of the use of Alf. Annex C provides a grammar for all of Alf, in a form better suited to automated processing than the productions used in the main body of the specification, which are intended for clarity of presentation rather than processing.

6.9 Acknowledgments

The following people from the various submitting organizations contributed to this specification:

- Ed Seidewitz, Model Driven Solutions
- Kim Letkeman, IBM
- Stephen Mellor, Mentor Graphics
- Manfred Koethe, 88 Solutions
- Nerijus Jankevicius, No Magic

We would also like to acknowledge Doug Tolbert, CSC, for his thorough review of the specification as the head of the Evaluation Working Group.

7 Lexical Structure

7.1 General

Lexically, an Alf input text can be considered to be a sequence of *input elements.* This clause describes the structure of these input elements. After lexical analysis, the text can then be interpreted as a sequence of *tokens* that are then parsed according to the Alf syntax, as defined in Clauses 8 through 10.

7.2 Line Terminators

The input text can be divided up into lines separated by *line terminators.* A line terminator may be a single character (such as a line feed) or a sequence of characters (such as a carriage return/line feed combination). This specification does not require any specific encoding for a line terminator, but any encoding used must be consistent throughout any specific input text. Any characters in the input text that are not a part of line terminators are referred to as *input characters.*

Grammar

```
LineTerminator
      = "\n"
InputCharacter
      = any character other than LineTerminator
```

7.3 Input Elements and Tokens

An *input element* can be white space, a lexical comment or a token. Tokens include documentation comments, names, reserved words, literals, punctuation and operators. After white space and lexical comments are removed, the sequence of tokens is interpreted according to the Alf syntax.

Grammar

```
InputText
      = InputElement { InputElement }
InputElement
      = WhiteSpace
      | LexicalComment
      | Token
Token = DocumentationComment
      | Name
      | ReservedWord
      | Literal
      | Punctuation
      | Operator
```

Cross References

1. WhiteSpace see 7.4
2. LexicalComment see 7.5.2
3. DocumentationComment see 7.5.3
4. Name see 7.6
5. ReservedWord see 7.7
6. Literal see 7.8

7. Punctuation see 7.9
8. Operator see 7.10

7.4 White Space

A *white space* character is a space, tab, form feed or line terminator. Any contiguous sequence of white space characters can be used to separate tokens that would otherwise be considered to be part of a single token. It is otherwise ignored.

There are two cases in which the line terminator is not syntactically considered to be white space. A list of annotations for a statement begins with the token "`//@`" and must end in a line terminator (see 9.2). And the heading for an in-line statement begins with the token "`/*@`" and must end in a line terminator (see 9.3).

Grammar

```
WhiteSpace
      = " " | "\t" | "\f"
      | LineTerminator
```

Cross References

1. LineTerminator see 7.2

7.5 Comments

7.5.1 General

Comments are used to annotate other elements of the input text. They have no computable semantics, but simply provide information useful to a human reader of the text. There are three kinds of comments:

1. An *end-of-line comment* includes all the text from the initial characters "`//`" to the end of the line, except that "`//@`" begins a statement annotation rather than a comment (see 9.2).
2. An *in-line comment* includes all the text from the initial characters "`/*`" to the final characters "`*/`", except that "`/**`" begins a documentation comment rather than a lexical comment (see below) and "`/*@`" begins an inline code block (see 9.2).
3. A *documentation comment* includes all the text from the initial characters "/**" to the final characters "*/". The *comment text* is the text *between* the initial characters "/**" and the final characters "*/".

The first two kinds of comments are together known as *lexical comments.*

7.5.2 Lexical Comments

Lexical comments are not considered tokens. Therefore they are stripped from the input text and not parsed as part of the Alf syntax. The comment symbols and all comment text are ignored. However, a comment cannot occur within a name (see 7.6) or a string literal (see 7.8.5).

Examples

```
// This is an end-of-line comment and will be ignored.
/* This is an in-line comment and will be ignored. */
```

Grammar

```
LexicalComment
      = EndOfLineComment
      | InLineComment
EndOfLineComment
      = "//" [ NotAt { InputCharacter } ] LineTerminator
NotAt = InputCharacter but not "@"
InLineComment
      = "/*" [ NotStarNotAt CommentText ] "*/"
CommentText
      = { NotStar } [ StarCommentText ]
StarCommentText
      = "*" [ NotStarNotSlash CommentText ]
NotStar
      = InputCharacter but not "*"
      | LineTerminator
NotStarNotAt
      = InputCharacter but not "*" or "@"
      | LineTerminator
NotStarNotSlash
      = InputCharacter but not "*" or "/"
      | LineTerminator
```

Cross References

1. InputCharacter see 7.2
2. LineTerminator see 7.2

7.5.3 Documentation Comments

Unlike lexical comments, documentation comments *are* lexically processed as tokens and can therefore be included as syntactic elements. The intent is for documentation comments to be mapped to UML comment elements, containing the comment text, that are actually included as part of the target model. Note that line terminators *are* allowed within documentation comments.

Examples

```
/** This is documentation text to be included in the model. */
```

Grammar

```
DocumentationComment
      = "/**" CommentText "*/"
```

Cross References

1. CommentText see 7.5.2

7.6 Names

The *name* of a named element denotes the element *without* reference to the namespace of which it is a member (if any). A name may contain any character. However, names that have the form of *identifiers* may be represented more simply.

The initial character of an identifier must be one of a lowercase letter, an uppercase letter or an underscore. The remaining characters of an identifier are allowed to be any character allowed as an initial character plus any digit. However, a *reserved word* may not be used as a name, even though it has

the form of an identifier (see 7.7). The Boolean literals `true` and `false` also have the form of identifiers, but they are considered lexically to be primitive literals rather than names (see 7.8.2).

An *unrestricted name,* on the other hand, is represented as a non-empty sequence of characters surrounded by single quotes. The characters within the single quotes may not include non-printable characters (including backspace, tab and newline). However, these characters may be included as part of the name itself through use of an *escape sequence.* In addition, the single quote character or the backslash character may only be included by using an escape sequence.

An escape sequence is a sequence of two text characters starting with the backslash as an *escape character,* which actually denotes only a single character (except for the newline escape sequence, which represents however many characters is necessary to represent an end of line in a specific implementation —see 7.2). Table 7.1 shows the meaning of the allowed escape sequences.

Table 7.1 Escape Sequences

Escape Sequence	Meaning
`\'`	Single Quote
`\"`	Double Quote
`\b`	Backspace
`\f`	Form Feed
`\t`	Tab
`\n`	Line Terminator
`\\`	Backslash

It is an error to follow a backslash in an unrestricted name with any other character than allowed in one of the escape sequences above.

Two names are the same if and only if they are composed of the same characters in the same sequence. In particular the case of alphabetic characters *is* significant (both in identifiers and unrestricted names).

However, an implementation is permitted to consider two names to be *conflicting* even if they are not the same, but every implementation must observe at least the following rules:

- Two different identifiers are never conflicting.
- The *corresponding identifier* for an unrestricted name is the identifier that results from prepending the name with an underscore and then removing all non-alphanumeric characters from the name other than underscore (_) and dollar sign (`$`). If the corresponding identifiers for two unrestricted names are different, then the original names do not conflict.

NOTE. The above implementation freedom for name conflicts is intended to make the handling of names simpler for implementations that map Alf text to a target language in which all names must follow the grammar of identifiers. The prepended underscore in a corresponding identifier assures that the result is a legal identifier even if the result would otherwise begin with a number or be empty. Dollar signs are included in corresponding identifiers because they are used in the standard names for equivalent bound elements—see 6.3.

Examples

```
customer
```

```
nextOrder
'+'
'orders in cart'
'On/Off Switch'
```

Grammar

```
Name  = Identifier
      | UnrestrictedName
Identifier
      = IdentifierChars but not a ReservedWord or BooleanLiteral
IdentifierChars
      = IdentifierLetter { IdentifierLetterOrDigit }
IdentifierLetterOrDigit
      = IdentifierLetter
      | Digit
IdentifierLetter
      = "a" … "z" | "A" … "Z" | "_"
Digit = "0"
      | NonzeroDigit
NonzeroDigit
      = "1" … "9"
UnrestrictedName
      = "'" NameCharacter { NameCharacter } "'"
NameCharacter
      = InputCharacter but not "'" or "\\"
      | EscapeSequence
EscapeSequence
      = "\\" EscapedCharacter
EscapedCharacter
      = "'" | "\"" | "b" | "f" | "t" | "n" | "\\"
```

Cross References

1. InputCharacter see 7.2

7.7 Reserved Words

A *reserved word* is a token that has the lexical structure of an identifier but is not allowed to actually be used as an identifier (see 7.6).

Grammar

```
ReservedWord
      = "abstract" | "accept" | "active" | "activity" | "allInstances" | "any"
      | "as" | "assoc" | "break" | "case" | "class" | "classify" | "clearAssoc"
      | "compose" | "createLink" | "datatype" | "default" | "destroyLink" | "do"
      | "else" | "enum"  | "for" | "from" | "hastype" | "if" | "import" | "in"
      | "inout" | "instanceof" | "let" | "namespace" | "new" | "nonunique" | "null"
      | "or" | "ordered" | "out" | "package" | "private" | "protected" | "public"
      | "receive" | "redefines" | "reduce" | "return" | "sequence" | "specializes"
      | "super" | "signal" | "switch" | "this" | "to" | "while"
```

7.8 Primitive Literals

7.8.1 General

A *primitive literal* is used to represent the value of a primitive type. Note that an enumeration literal is not actually denoted as a literal in a lexical sense, but rather by its name as a named element (see also 8.3.3 on name expressions).

Grammar

```
PrimitiveLiteral
      = BooleanLiteral
      | NaturalLiteral
      | UnboundedValueLiteral
      | StringLiteral
      | RealLiteral
```

Cross References

1. BooleanLiteral see 7.8.2
2. NaturalLiteral see 7.8.3
3. UnboundedValueLiteral see 7.8.4
4. StringLiteral see 7.8.5
5. RealLiteral see 7.8.6

7.8.2 Boolean Literals

A *Boolean literal* represents a literal Boolean model element, with the primitive type `Boolean`. The literal "`true`" represents an element with the value true, while the literal "`false`" represents an element with the value false.

Grammar

```
BooleanLiteral
      = "true" | "false"
```

7.8.3 Natural Literals

A *natural literal* represents a natural number—that is, a non-negative integer. The sets of values of the primitive types `Integer` and `UnlimitedNatural` both have the natural numbers as a subset. A natural literal may thus be used to represent values of either of these types.

NOTE. An effective literal for negative values of type `Integer` can be obtained by applying the unary numeric negation operator (see 8.5.4) to a natural literal. The unbounded value of type `UnlimitedNatural` has its own literal (see 7.8.4).

The Alf standard model library `PrimitiveTypes` package includes the type `Natural` that is a specialization of both `Integer` and `UnlimitedNatural` (see 11.3.1), and natural literals are considered to inherently have this type. However, since UML does not provide any literal specification metamodel representation for `Natural`, any time a natural literal is used, it must be possible to determine from context whether it is an integer or an unlimited natural value that is actually required, so the proper metamodel representation can be chosen when the literal is mapped to fUML. If this cannot be determined implicitly, then an explicit cast (see 8.5.5) to type `Integer` or type `UnlimitedNatural` must be used.

A natural literal may be expressed in decimal (base 10), binary (base 2), octal (base 8) or hexadecimal (base 16). A decimal literal consists of either the single character "`0`", representing the integer 0, or a digit from "`1`" to "`9`" optionally followed by one or more digits from "`0`" to "`9`", representing a positive integer. A binary literal consists of the prefix "`0b`" or "`0B`" followed by one or more of the binary digits "`0`" or "`1`". A hexadecimal literal consists of the prefix "`0x`" or "`0X`" followed by one or more hexadecimal digits. Hexadecimal digits with values 10 through 15 are represented by the letters "`a`" through "`f`" or "`A`" through "`F`", respectively (case is not significant). An octal literal consists of the digit "`0`" followed by one or more digits from "`0`" to "`7`". Underscores may be inserted between digits but are ignored in determining the value of the literal.

Subclause 9.3.2 of the fUML Specification allows a conforming implementation to limit the supported values for `Integer` and `UnlimitedNatural` types to a finite set. An Alf implementation may also adopt such a limitation, in which case it may reject any natural literal representing a value outside the supported set.

Examples

```
1234

0

0b1010111000010000

0B0100_1010_0101_1011

0xAE10

0X4a_5b

057410

0_045_133
```

Grammar

```
NaturalLiteral
      = DecimalLiteral
      | BinaryLiteral
      | HexLiteral
      | OctalLiteral
DecimalLiteral
      = "0"
      | NonzeroDigit { [ "_" ] Digit }
BinaryLiteral
      = BinaryPrefix BinaryDigit { [ "_" ] BinaryDigit }
BinaryPrefix
      = "0" "b"
      | "0" "B"
BinaryDigit
      = "0" | "1"
HexLiteral
      = HexPrefix HexDigit { [ "_" ] HexDigit }
HexPrefix
      = "0" "x"
      | "0" "X"
HexDigit
      = Digit
      | "a" … "f"
      | "A" … "F"
```

```
OctalLiteral
      = "0" [ "_" ] OctalDigit { [ "_" ] OctalDigit }
OctalDigit
      = "0" … "7"
```

Cross References

1. NonzeroDigit see 7.6
2. Digit see 7.6

7.8.4 Unbounded Value Literals

An *unbounded value literal* represents a literal unlimited natural model element for the value *unbounded* of the primitive type `UnlimitedNatural`. Other unlimited natural values are represented as natural literals (see 7.8.3).

Grammar

```
UnboundedValueLiteral
      = "*"
```

7.8.5 String Literals

A *string literal* represents a literal string model element, with the primitive type `String`. The string value of the element is given as a sequence of characters, with escape characters resolving to their meaning as given in 7.6, surrounded by double quote characters (which are not included as part of the string value). The empty string is represented by a pair of double quote characters with no other characters intervening between them.

Examples

```
"This is a string."
```

```
"This is a string with a quote character (\") in it."
```

```
"This is a string with a new line (\n) in it."
```

Grammar

```
StringLiteral
      = "\"" { StringCharacter } "\""
StringCharacter
      = InputCharacter but not "\"" or "\\"
      | EscapeSequence
```

Cross References

1. InputCharacter see 7.3
2. EscapeSequence see 7.6

7.8.6 Real Literals

A *real literal* represents a real number, with the primitive type `Real`. A real literal consists of a decimal (base 10) whole-number part, optionally followed by a decimal point and a decimal fraction part, optionally followed by an exponent (power of 10) part. If a real literal includes an exponent, then the decimal point is optional; otherwise it is required (even if not followed by a fraction part). An exponent part is indicated by the letter "`e`" or "`E`", followed by an optionally signed integer. Underscores may be

inserted between digits in the whole-number, fraction or exponent parts, but are ignored in determining the value of the literal.

Subclause 9.3.3 of the fUML Specification allows a conforming implementation to limit the range of the supported values for type `Real`. An Alf implementation may also adopt such a limitation, in which case it may reject any real literal representing a value outside the supported range. The fUML Specification also allows a conforming implementation to only support a restricted value set for type `Real`. An Alf implementation may also adopt such a limitation, in which case it may map the value of a real literal to the closest value supported within the implemented value set. Finally, the fUML Specification allows a conforming implementation to include additional special values that are instances of the `Real` type but are not numeric values (such as infinite values and "not a number" values used in some floating-point implementations). Alf does not provide real literal representations for such values.

Examples

```
3.14

0.0

1234.

.0625

5E3

0.314e+1

2E-10

6.022_140_9e+23

0.1E1_000
```

Grammar

```
RealLiteral
      = [ DecimalNumeral ] "." DecimalNumeral [ ExponentPart ]
      | DecimalNumeral ExponentPart

DecimalNumeral
      = Digit { [ "_" ] Digit }

ExponentPart
      = ExponentIndicator [ Sign ] DecimalNumeral

ExponentIndicator
      = "e" | "E"

Sign = "+" | "-"
```

Cross References

1. Digit see 7.6

7.9 Punctuation

The tokens below are considered to be *punctuation.*

NOTE. Some tokens below are made up of two symbols that may themselves individually be tokens. Nevertheless, the two-symbol token is not considered a combination of the individual symbol tokens. For example, "`::`" is considered a single token, not a combination of two "`:`" tokens. Input characters

are grouped from left to right to form the longest possible sequence of characters to be grouped into a single token. Thus “`a::b`” would analyzed into four tokens: “`a`”, “`::`”, “`:`” and “`b`”.

Grammar

```
Punctuation
      = "(" | ")"  | "{" | "}"  | "["  | "]" | ";" | "," | "." | ":"
      | ".." | "::" | "=>" | "->"
```

7.10 Operators

The tokens below are considered to be *operators*.

Grammar

```
Operator
      = "="  | ">"  | "<"  | "!"  | "~"  | "?"  | "@"  | "$"
      | "==" | "<=" | ">=" | "!=" | "&&" | "||" | "++" | "--"
      | "+"  | "-"  | "*"  | "/"  | "&"  | "|"  | "^"  | "%"
      | "+=" | "-=" | "*=" | "/=" | "&=" | "|=" | "^=" | "%="
      | "<<" | ">>" | ">>>" | "<<=" | ">>=" | ">>>="
```

8 Expressions

8.1 Overview

An *expression* is a behavioral unit that *evaluates* to a (possibly empty) collection of values. Expressions may also have side effects, such as changing the value of an attribute of an object.

The *full conformance* level includes all kinds of expressions specified in this clause. However, the *minimum conformance* level only requires a subset of the full expression syntax. Therefore, in each of the concrete syntax grammar productions given in the subclauses of this clause, some portion of the production may be italicized. Only the italicized portions apply at the minimum conformance level. Unitalicized portions may be ignored for minimum conformance. (See also 2.2 on the definition of syntactic conformance.)

This subclause describes the top-level syntax and semantics for expressions. The next level categorization of expressions syntactically is into conditional-test expressions (see 8.7) and assignment expressions (see 8.8). However, in the subclauses following this subclause, expressions are described in a traditional "bottom up" fashion, starting with the simplest forms of expressions and working back up to conditional-test and assignment expressions.

Syntax

```
Expression(e: Expression)
      = ConditionalExpression(e)
      | AssignmentExpression(e)
```

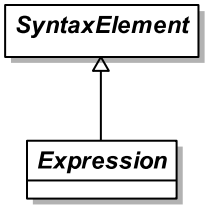

Figure 8.1 Base Abstract Syntax for Expressions

Cross References

1. Syntax Element see 6.6
2. ConditionalExpression see 8.7
3. AssignmentExpression see 8.8

Semantics

Integration with UML

An Alf expression can be inserted into a UML model using an *opaque expression* (UML Specification, 7.3.35) in which the unprocessed text of the Alf expression is used as the body of the opaque expression and the corresponding language string is `"Alf"`. Opaque expressions are kinds of *value specifications* (UML Specification, 7.3.54). Thus, an Alf expression may be used in a UML model any place that a value specification is allowed.

In addition, a special form of *initialization expression* may be for instance creation and sequence construction expressions acting as the specification of default values of properties or parameters. In such initialization expressions, an explicit description of the type of the expression does not need to be included, as is the case for normal instance creation and sequence construction expressions, since this can be inferred from the declared type of the property or parameter. (Instance creation expressions are described in 8.3.12 and sequence construction expressions are described in 8.3.15. The use of their special forms in initialization descriptions within Alf is discussed for local name declaration statements in 9.6 and for property initialization in 10.5.2.)

The execution semantics of an Alf expression are given formally by the mapping to UML activity graphs given in the following subclauses. An Alf expression can therefore always be *compiled* to a part or all of a UML activity model (which does not necessarily need to be the same as the formal mapping, but must have an equivalent effect to it—see 2.3). If the expression appears as part of a statement, then the compilation of the expression will be part of the compilation of the statement (see Clause 9). Otherwise, the compilation of the expression may be inserted as an activity that is the associated behavior of a corresponding opaque expression (see UML Superstructure 13.3.21) that is constructed as follows:

- A single return parameter of the type of the Alf expression.
- The activity nodes and edges resulting from the compilation of the Alf expression (semantically equivalent to the formal mapping as specified in the following subclauses).
- A single activity parameter node associated with the return parameter, which becomes the target of an outgoing object flow from the *result source* element from the mapping of the Alf expression (see the definition of "result source element" given under Mapping below).

Indeed, the semantics of an opaque expression that only includes unprocessed Alf text in its body may be considered equivalent to an opaque expression with an associated behavior constructed strictly according to the formal fUML mapping for the Alf expression.

NOTE. Opaque expressions are not included at all in fUML, so the execution of such an expression, even one with a behavior conformant to the Alf subset, is still not fully defined within by the fUML standard.

An expression either has a statically determined *type* or can be statically determined to be untyped. All values resulting from evaluation of the expression will be of the type of the expression, if it has one. If the expression is untyped, then the result values may have any type. For simplicity of terminology in this clause, an untyped expression will be referred to as having "type `any`".

An expression also has a statically determined *multiplicity.* The number of values resulting from the evaluation of the expression will be no smaller than the multiplicity lower bound and no higher than the multiplicity upper bound (or unbounded if the upper bound is the unbounded value "`*`").

Local Names and Assigned Sources

The evaluation of an expression may depend on the values assigned to *local names* used in the expression. Local names are used in Alf to denote intermediate values in computations within a statement sequence (see 9.1). Alf is designed so, on the surface, local names can be assigned and reassigned in a similar way to variables in more traditional programming languages. However, the underlying fUML metamodel for actions and activities is fundamentally based on data flow, not on an implicit underlying store of variables. Therefore, the assignments of and references to local names in an Alf input text need to be mapped to appropriate object flow edges from the mapping of the appropriate assignment to the mapping of the reference that requires that assigned value.

Carrying out this mapping requires an analysis of the set of local names that are *statically* known to have assigned values during the execution of any Alf statement. The *assigned source* for a local name is the syntax element that, when executed, will provide the actual *assigned value* for that local name. If the assigned source for a local name is known, then a reference to the assigned value of that local name can be mapped to an object flow from the mapping of the assigned source. (See also the discussion of local names relative to statements in Clause 9.)

Since a local name may also be defined or reassigned within an expression, one can refer to the *assignments* (i.e., the statically determined assigned sources) for local names both *before* and *after* the evaluation of an expression. Most kinds of expressions do not actually themselves change the assigned source of a name. An assignment expression (see 8.8) is, of course, the main kind of expression that changes the assigned source for a name. However, invocation expressions (see 8.3.7) may assign names via `out` and `inout` parameters. Increment and decrement expressions also act as assignments (see 8.4).

Other than the kinds of expressions listed above, the only way that an assigned source may change in an expression is if it contains one of the above kinds of expression, directly or indirectly, as a subexpression. Unless otherwise stated, it can be assumed that the assigned source of a name before any subexpression of an expression is the same as the assigned source before the expression.

Further, in general, it is only legal to assign a name in at most one subexpression of any expression. Therefore, unless otherwise stated, the assigned source for a name after an expression is the same as after any subexpression. Specific assignment rules for various kinds of expressions are described in their respective subclauses.

NOTE. The above rule allows most subexpressions to be evaluated concurrently.

8.2 Qualified Names

A *name* is used to identify a UML *named element* (see UML Superstructure, 7.3.34), which may or may not be a member of one or more *namespaces* (see UML Superstructure, 7.3.35). A named element that is not a member of any namespace is referred to as a *local name,* and the scope within which it can be referenced is limited. In contrast, a named element that is a member of a namespace may be referenced from outside the namespace in which it is defined (depending on its visibility – see UML Superstructure, 7.3.56) and, if defined in a package (see UML Superstructure, 7.3.38), may be imported into another namespace (see UML Superstructure, subclauses 7.3.15 and 7.3.40).

A namespace is itself a named element. A *qualified name* is one that includes both the unqualified name of a named element as well as the name of a namespace of which the named element is a member. The name of the namespace may or may not itself be qualified. A local name is never qualified.

A qualified name has the form of a list of the names of namespaces followed by the unqualified name of the named element. Syntactically, the names in the list are separated by the either the symbol "`::`" or the symbol "`.`"

NOTE. The UML Superstructure specifies the symbol "`::`" as the separator used in qualified names. However, it is also common in other languages to use "`.`" in qualified names, so Alf allows either. Only one or the other must be used throughout a single qualified name, though.

If any individual name listed in a qualified name is for a *template* (see 6.3 on templates), then a *template binding* may be optionally provided with that name. A template binding lists the qualified names of argument elements to be substituted for each of the formal template parameters in the template, surrounded by the angle brackets "<" and ">".

The *fully qualified name* of a named element is either its unqualified name, if it is not owned by a namespace, or else its name qualified with the fully qualified name of its owning namespace.

Examples

```
customer

Ordering::Order::customer

Ordering.Order.customer

FoundationalModelLibrary::BasicInputOutput

FoundationalModelLibrary.BasicInputOutput

Set<Integer>

Map<K=>String, V=>Entry>

Map<String,Entry>.KeySet

List< List<String> >

CollectionClasses::Set<Integer>::add
```

Syntax

```
TypeName(q: QualifiedName)
      = QualifiedName(q)
      | "any"
QualifiedName(q: QualifiedName)
      = ColonQualifiedName(q)
      | DotQualifiedName(q)
      | UnqualifiedName(q)
PotentiallyAmbiguousQualifiedName(q: QualifiedName)
      = ColonQualifiedName(q)
      | DotQualifiedName(q) (q.isAmbiguous=true)
      | UnqualifiedName(q)
ColonQualifiedName(q: QualifiedName)
      = NameBinding(q.nameBinding) "::" { NameBinding(q.nameBinding) "::" }
        NameBinding(q.nameBinding)
DotQualifiedName(q: QualifiedName)
      = NameBinding(q.nameBinding)) "." { NameBinding(q.nameBinding) "." }
        NameBinding(q.nameBinding)
UnqualifiedName(q: QualifiedName)
      = NameBinding(q.nameBinding)
NameBinding(n: NameBinding)
      = Name(n.name) [ TemplateBinding(n.binding) ]
TemplateBinding(b: TemplateBinding)
      = PositionalTemplateBinding(b)
      | NamedTemplateBinding(b)
PositionalTemplateBinding(b: PositionalTemplateBinding)
      = "<" QualifiedName(b.argumentName)
        { "," QualifiedName(b.argumentName) } ">"
NamedTemplateBinding(b: NamedTemplateBinding)
      = "<" TemplateParameterSubstitution(b.substitution)
        { "," TemplateParameterSubstitution(b.substitution) } ">"
TemplateParameterSubstitution(s: TemplateParameterSubstitution)
      = Name(s.parameterName) "=>" QualifiedName(s.argumentName)
```

NOTE. Named template binding notation is not available at the minimum conformance level (see 2.2).

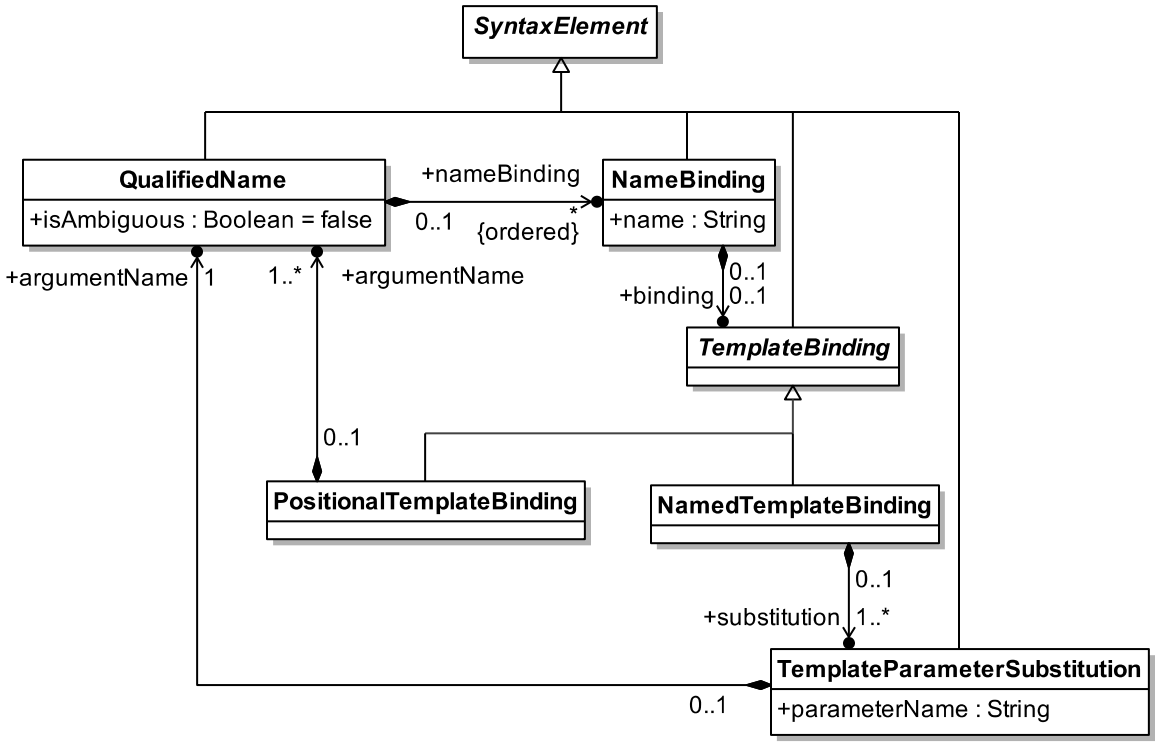

Figure 8.2 Abstract Syntax of Qualified Names

Cross References

1. Syntax Element see 6.6
2. Name see 7.6

Disambiguation

It is ambiguous whether a syntactic element of the form `n₁.n₂. … .nₘ`, where the `nᵢ` are name bindings and $m \geq 2$, should be parsed as a qualified name or a feature reference (see 8.3.6) when the element is used in the following contexts.

- As an `Expression` (`NameExpression`, 8.3.3, versus `PropertyAccessExpression`, 8.3.6)
- As an `InvocationTarget` (`BehaviorInvocationTarget`, 8.3.9, versus `FeatureInvocationTarget`, 8.3.10)
- As a `LeftHandSide` (`QualifiedName` versus `FeatureReference`, see 8.8)

In these cases, the element is initially parsed as a qualified name with `isAmbiguous=true`. It is then disambiguated as follows:

- If $n_1.n_2. \ldots .n_{m-1}$ resolves to a namespace, then the original element should be considered a qualified name.
- Else, the original element should be considered a feature reference, with name n_m and target expression determined by the disambiguation (if necessary) of $n_1.n_2. \ldots .n_{m-1}$.

Semantics

Element naming and namespaces do not have executable semantics and are therefore not addressed in the fUML Specification. Nevertheless, the use of named elements is particularly important in Alf, since, in a textual notation, model elements are primarily referenced by name. Such named references must be *resolved* into actual abstract syntax element references when the Alf text is mapped into a fUML model.

The semantics of names, namespaces and visibility are defined in detail in the UML Superstructure, particularly in subclauses 7.3.35 Namespace, 7.3.56 VisibilityKind, 7.3.15 ElementImport and 7.3.40 PackageImport. Depending on the context in which it is used (as defined in subsequent subclauses), a name may either be part of the definition of a named element or it may denote a reference to a named element defined elsewhere.

Current Scope

The *current scope* for the resolution of a reference to a name is the specific innermost namespace in which that named reference lexically appears.

A name is said to be *visible in the current scope* if a named element with the given name is a member of the current scope namespace or is visible in the namespace immediately enclosing the current scope (if any). Such an element may be referenced using an *unqualified* name.

Otherwise the first name listed in a qualified name must be visible as an unqualified name in the *current scope* in which the qualified name occurs. Each succeeding name must be the name of a visible member of the preceding namespace.

Type Names

A *type* constrains the values represented by a *typed element* (see UML Superstructure, 7.3.52). Any value in UML is an instance of some *classifier,* so a type is always given by naming a classifier, which constrains the typed element to instances of that classifier. If the named classifier is a *template* (see 6.3 on templates), then an argument type must be given for each parameter of the template. The association of a classifier template with its arguments is known as a *type binding.*

A type name may be either a qualified name or the keyword `any`. If a type name is a qualified name, then this name must resolve to a classifier, which constrains a typed element to represent the values that are instances of that classifier. The keyword `any` is used to indicate that a typed element is actually *untyped,* that is, that there are no constraints on the values it may represent.

Type Conformance

One classifier *conforms* to another if the two classifiers are equivalent or if any direct generalization of the first classifier conforms to the second classifier. For the purpose of conformance, two classifiers are considered equivalent if they are the same or if they both have template bindings for equivalent templates with equivalent arguments for all template parameters (for non-classifier template parameters, the arguments must be identical).

NOTE. Type conformance as defined in UML Superstructure, 7.3.8, does not include the above rule for equivalence of classifiers with template bindings. This additional rule follows from the substitution semantics used in Alf for template bindings, such that classifiers with equivalent bindings are considered to have the same equivalent bound element after substitution of actual parameters (see 6.3).

Template Bindings

If a qualified name resolves to a *template*—that is, an element with *template parameters*—then a *template binding* may be appended to the qualified name. Such a binding names an argument element to

be substituted for each template parameter. Such a qualified name with a template binding may itself be used as the qualification part of a larger qualified name.

The argument elements in a template binding may not themselves be templates. Each argument must be *compatible* with the corresponding template parameter. In general, a template parameter may represent any kind of packageable element, any kind of connectable element or an operation, and a compatible argument for a template parameter must be an element of the same kind. In addition, the following special compatibility rules must hold for specific kinds of elements:

- *Classifier.* A classifier template parameter may have *constraining classifiers.* A classifier template parameter with no constraining classifiers may be substituted with any classifier. A template parameter with constraining classifiers must be substituted with an argument that conforms to *all* of the constraining classifiers (see the definition of type compatibility above).
- *Value Specification.* An argument is compatible with a value specification template parameter if it is a value specification whose type conforms to the type of the value specification represented by the template parameter.
- *Operation.* An argument is compatible with an operation template parameter if it is an operation with the same number of parameters, in the same order, with the same types as the operation represented by the template parameter.
- *Connectable Element.* An argument is compatible with a connectable element template parameter if it is the same kind of connectable element with the same type as the connectable element represented by the template parameter.

NOTE. Templates are specified in 17.5 of the UML Superstructure. Specifically:

- Classifiers as parameterable elements are described in UML Superstructure, 17.5.7, and classifier template parameters are described in 17.5.8. The above rule for compatibility presumes that allowSubstitutable=false for all classifier template parameters with constraining classifiers.
- Value specifications as parameterable elements are described in UML Superstructure, 17.5.20, including the compatibility rule given above.
- Operations as parameterable elements are described in UML Superstructure, 17.5.15, and operation template parameters are described in 17.5.16. However, even though 17.5.16 mentions “additional semantics related to the compatibility of actual and formal operation parameters”, no such additional semantics are actually provided in 17.5.15. Nevertheless, the compatibility rule given above for operations given above is necessary to ensure that the substitution of an argument for an operation template parameter leaves the model well formed.
- Connectable elements as parameterable elements are described UML Superstructure, 17.5.17, but this subclause does not specify any special compatibility rule for connectable elements. However, the compatibility rule given above for connectable elements is necessary in general to ensure that the substitution of an argument for a connectable element template parameter leaves the model well formed. UML Superstructure, 17.5.19, also describes specifically properties as parameterable elements, giving a compatibility rule of type conformance that is looser than the rule for connectable elements given above. However, since a property could be used within the body of a template as both a value and the target of an assignment, simple type conformance is not enough to ensure well-formedness on substitution. Instead, the arguments for template parameters representing properties (which are, in the end, kinds of connectable elements) are required in Alf to follow the general connectable element compatibility rule given above.

Either positional or named notation may be used for template arguments. If positional notation is used, then the template arguments are matched to corresponding template parameters in order, and arguments must be provided for all template parameters. If named notation is used, then each template parameter must be named in exactly one template parameter substitution, but the order of the substitutions is irrelevant.

For example, the standard library class `Set` has a single template parameter `T`. The following are all legal bindings for this template, assuming that `Task` names a classifier.

```
Set<Integer>
```

```
Set<Task>
```

```
Set<Set<Integer> >
```

NOTE. The space between the right angle brackets at the end of the last example above are necessary in order for it to parse correctly. If there was no space, then the symbol “>>” would be recognized as a right shift operator, not two angle brackets (see 7.9 on the lexical analysis of multi-character symbols).

The standard library class `Map` has two template parameters, `Key` and `Value`. The following are all equivalent bindings for this template.

```
Map<String, Definition>
```

```
Map<Key=>String, Value=>Definition>
```

```
Map<Value=>Definition, Key=>String>
```

8.3 Primary Expressions

8.3.1 Overview

Primary expressions include the simplest kinds of expressions, from which more complicated kinds are constructed. Parenthesized expressions are also considered primary expressions. The categorization into primary expressions is purely a concept of the concrete syntax, with no additional abstract syntax or mapping specification beyond that given for the various kinds of expressions so categorized.

Syntax

```
PrimaryExpression(e: Expression)
      = NameExpression(e)
      | NonNamePrimaryExpression(e)
NonNamePrimaryExpression(e: Expression)
      = LiteralExpression(e)
      | ThisExpression(e)
      | ParenthesizedExpression(e)
      | PropertyAccessExpression(e)
      | InvocationExpression(e)
      | InstanceCreationExpression(e)
      | LinkOperationExpression(e)
      | ClassExtentExpression(e)
      | SequenceConstructionExpression(e)
      | SequenceAccessExpression(e)
      | SequenceOperationExpression(e)
      | SequenceReductionExpression(e)
      | SequenceExpansionExpression(e)
```

Cross References

1. LiteralExpression see 8.3.2
2. NameExpression see 8.3.3
3. ThisExpression see 8.3.4
4. ParenthesizedExpression see 8.3.5
5. PropertyAccessExpression see 8.3.6
6. InvocationExpression see 8.3.7
7. InstanceCreationExpression see 8.3.12
8. LinkOperationExpression see 8.3.13
9. ClassExtentExpression see 8.3.14
10. SequenceConstructionExpression see 8.3.15
11. SequenceAccessExpression see 8.3.16
12. SequenceOperationExpression see 8.3.17
13. SequenceReductionExpression see 8.3.18
14. SequenceExpansionExpression see 8.3.19

Semantics

See the discussion of semantics for each kind of expression in subsequent subclauses.

8.3.2 Literal Expressions

A *literal expression* comprises a single primitive literal (see 7.8). (Note that enumeration literals are not denoted using literal expression but, rather, using name expressions—see 8.3.3.)

Syntax

```
LiteralExpression(e: LiteralExpression)
      = BooleanLiteralExpression(e)
      | NaturalLiteralExpression(e)
      | UnboundedLiteralExpression(e)
      | StringLiteralExpression(e)
      | RealLiteralExpression(e)
BooleanLiteralExpression(e: BooleanLiteralExpression)
      = BooleanLiteral(e.image)
NaturalLiteralExpression(e: NaturalLiteralExpression)
      = NaturalLiteral(e.image)
UnboundedLiteralExpression(e: UnboundedLiteralExpression)
      = UnboundedValueLiteral
StringLiteralExpression(e: StringLiteralExpression)
      = StringLiteral(e.image)
RealLiteralExpression(e: RealLiteralExpression)
      = RealLiteral(e.image)
```

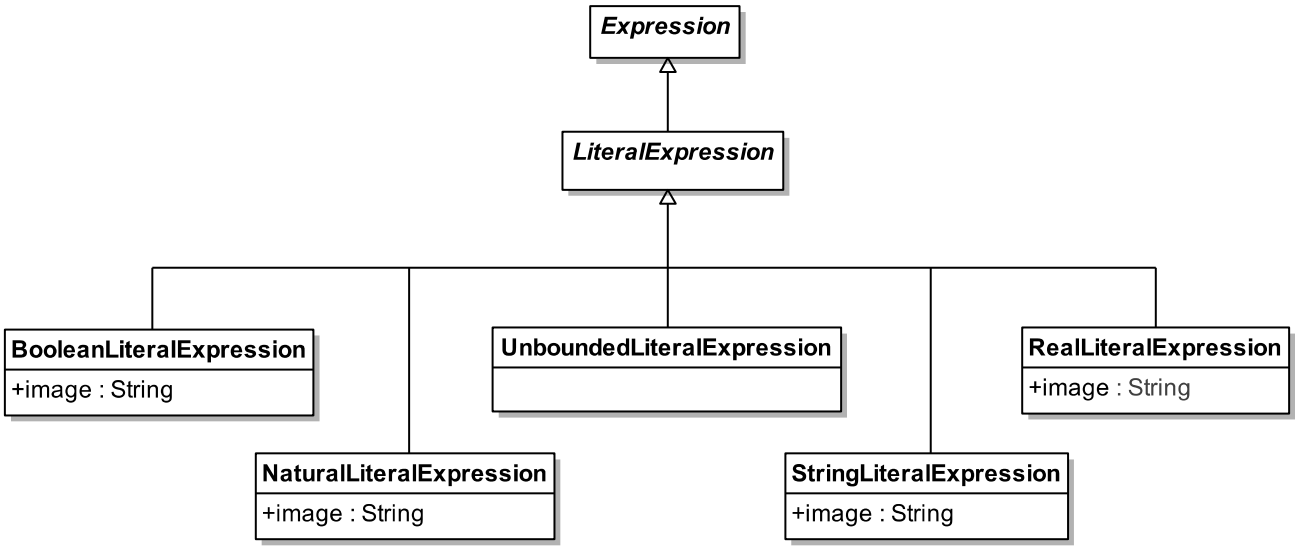

Figure 8.3 Abstract Syntax of Literal Expressions

Cross References

1. BooleanLiteral see 7.8.2
2. NaturalLiteral see 7.8.3
3. UnboundedValueLiteral see 7.8.4
4. StringLiteral see 7.8.5
5. RealLiteral see 7.8.6
6. Expression see 8.1

Semantics

A literal expression evaluates to the single primitive value denoted by its primitive literal.

The type of a literal expression is the primitive type corresponding to the kind of primitive literal it comprises. Its multiplicity is always `[1..1]`.

8.3.3 Name Expressions

A *name expression* evaluates to the values denoted by a name.

Syntax

```
NameExpression(e: NameExpression)
      = PotentiallyAmbiguousQualifiedName(e.name)
```

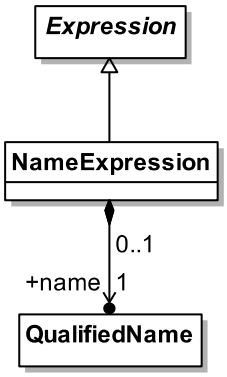

Figure 8.4 Abstract Syntax for Name Expressions

Cross References

1. Expression see 8.1
2. QualifiedName see 8.2
3. PotentiallyAmbiguousQualifiedName see 8.2

NOTE. see 8.2 for rules on the disambiguation of a qualified name with the dot notation used as a name expression versus a property access expression.

Semantics

The given name for a name expression must either resolve to an enumeration literal name or to a parameter or local name with an assigned source before the name expression. However, a name with a `@parallel` annotation as its assigned source may not be used as a name expression (as discussed in 9.12).

A local name is never qualified.

A parameter name may only be used if the name expression appears within the body of a behavior with parameters (see UML Superstructure, 13.3.2). The parameter name may be qualified by the name of the behavior or, if the behavior is the method of an operation, by the name of the operation. But this is never required since the names of the parameters of a behavior are always in the current scope of the definition of the behavior.

A name expression is evaluated as follows:

- *Local Name.* The values of a local name are those given by its assigned source.
- *Parameter Name.* The values of a parameter name are those given by its assigned source. For an `in` or `inout` parameter, this will initially be the value assigned to the parameter when the enclosing behavior began execution. However, an `inout` parameter may be reassigned, in which case its assigned source will change (see 8.8). An `out` parameter is always given a value only by explicit assignment (see 8.8).
- *Enumeration Literal Name.* The value of an enumeration literal name is the named enumeration literal.

The type and multiplicity of a name expression are the same as its name, determined as given below:

- *Local Name.* As determined by its assigned source (see 8.8).

- *Parameter Name.* For an `in` parameter or and `inout` parameter that has not been assigned, as declared for the named parameter. Otherwise, as determined for its assigned source (see 8.8).
- *Enumeration Literal.* The type is the corresponding enumeration and the multiplicity is `[1..1]`.

8.3.4 `this` Expressions

A *this expression* consists of the keyword `this`. It evaluates to a reference to the context value for the context in which the `this` expression occurs.

Syntax

```
ThisExpression(e: ThisExpression)
      = "this"
```

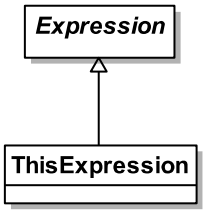

Figure 8.5 Abstract Syntax of `this` Expressions

Cross References

1. Expression see 8.1

Semantics

The static type of a `this` expression is the statically determined *context classifier* for the context in which the `this` expression occurs. The context classifier is determined as follows:

- If the expression appears in a method, classifier behavior or property default value, the context classifier is the classifier that owns the method, classifier behavior or property.
- If the expression appears in a behavior (other than a classifier behavior) that is owned by another behavior, then the context classifier is the context of the owning behavior. (For example, if a state machine is acting as a classifier behavior, then its context classifier is the classifier that owns it, and this is also the context classifier for all effect, entry, exit and do behaviors within it.)
- Otherwise the context classifier is the behavior containing the `this` expression.

NOTE. The derivation of the context property of a behavior is defined in 13.3.2 of the UML Superstructure.

The context value to which a `this` expression evaluates is determined as follows:

- For the method of a behavioral feature (see UML Superstructure, 13.3.3) other than a constructor, the context value is the value on which the behavioral feature was invoked.
- For a constructor (see UML Superstructure, 9.3.1) or property default value (see UML Superstructure, 7.3.44), the context object is the newly constructed value.
- For a classifier behavior (see UML Superstructure, 13.3.4), the context value is the instance of the active class for which the behavior is executing.

- For a behavior owned by another behavior (such as an entry action on a state machine), the context value is the context value of the invoking instance owning behavior.
- Otherwise, the context value is the behavior instance being executed.

Note that the dynamic type of the context value returned by a `this` expression may actually be a subclass of the static type of the `this` expression.

The multiplicity of a `this` expression is `[1..1]`.

8.3.5 Parenthesized Expressions

A *parenthesized expression* is a contained expression surrounded by parentheses.

Syntax

```
ParenthesizedExpression(e: Expression)
      = "(" Expression(e) ")"
```

NOTE. A parenthesized expression has the abstract syntax of its contained expression.

Semantics

A parenthesized expression is evaluated by evaluating the contained expression and results in the values of the contained expression. The use of parenthesizes only effects order of evaluation, not how the contained expression is evaluated.

The type and multiplicity of a parenthesized expression are the same as the contained expression.

8.3.6 Property Access Expressions

A *property access expression* is used to access the value of a property of instances of a classifier. It is denoted by *feature reference,* which consists of a target primary expression (see 8.3.1) and the name of a property of the type of the target expression.

Examples

```
poleValue.im
```

```
this.node
```

```
members.name
```

```
jack.house
```

Concrete Syntax

```
PropertyAccessExpression(e: PropertyAccessExpression)
      = FeatureReference(e.featureReference)
FeatureReference(f: FeatureReference)
      = FeatureTargetExpression(f.expression) "." NameBinding(f.nameBinding)
FeatureTargetExpression(e: Expression)
      = NameTargetExpression(e)
      | NonNamePrimaryExpression(e)
NameTargetExpression(e: NameExpression)
      = ColonQualifiedName(e.name)
```

NOTE. see 8.2 for rules on the disambiguation of a qualified name using the dot notation to a property access expression. Such a potentially ambiguous expression is always initially parsed as a qualified name. This is why a NameTargetExpression is not allowed to be a DotQualifiedName or an

UnqualifiedName, since, along with the dot notation for the feature reference, this should initially be parsed as a potentially ambiguous qualified name rather than a feature reference.

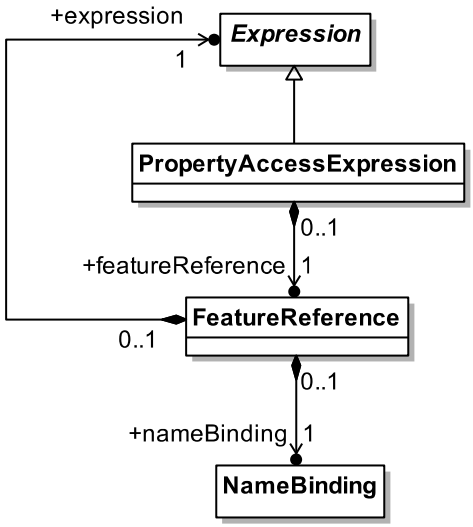

Figure 8.6 Abstract Syntax of Property Access Expressions

Cross References

1. Name see 7.6
2. Expression see 8.1
3. ColonQualifiedName see 8.2
4. NameBinding see 8.2
5. NonNamePrimaryExpression see 8.3.1

Semantics

The target expression in a property access expression may not be untyped nor may its type be a primitive type or enumeration. The identified property name must denote either:

- A structural feature (owned or inherited) of the type of the target expression.
- The name of an association end of a binary association, the opposite end of which is typed by the type of the target expression.

If the identified property is a template, then a template binding must be provided with arguments for all its template parameters.

For it to be legal to use the name of an association end in a property access expression, there must be exactly one binary association visible in the current scope that meets the above criterion and the given name must not also be the name of a structural feature of the type of the target expression.

If the target expression has multiplicity `[1..1]`, then result of the collection expression will always be a single instance. This is known as a *single instance property access.* If the target expression has multiplicity other than `[1..1]`, then the property access expression is known as a *sequence property access.* Such an expression is equivalent to a `collect` expression (see 8.3.21) as described below under Sequence Property Access.

The type of a property access expression is the same as the type of the named property. The multiplicity upper and lower bounds of the property access expression are equal to the product of the upper and lower bounds, respectively, of the named property and the target expression.

A property access expression is evaluated by first evaluating the target expression, which results in a sequence of instances. The result of the property access expression is then a sequence containing the union of the values of the named property for each of the target instances.

Single Instance Property Access

In the case of a single instance property access expression, the target expression will always evaluate to exactly one instance. If the property name is for a structural feature, then the resulting values of the property access expression are the values of that structural feature for the given instance.

A property access expression may also be used to access the values of an opposite association end of a binary association in which the instance participates. In this case, the resulting values are the values of named end of all links of the association for which the value of the opposite end is the given instance.

As an example of an association end access, consider the following association (represented in Alf notation—see 10.4.5).

```
assoc Owns {
  owner: Person;
  house: House[*];
}
```

If the association `Owns` is in the current scope (that is, visible without qualification), and `jack` is a `Person`, then the expression

```
jack.house
```

is equivalent to the association read expression (see 8.3.9)

```
Owns::house(owner => jack)
```

Sequence Property Access

A sequence property access expression of the form *`primary.name`* is equivalent to a `collect` expression (see 8.3.21) of the form

```
primary -> collect x (x.name)
```

NOTE. It is *not* an error for the result of the target expression in a property access expression to be empty. In this case, the property access expression evaluates to an empty sequence.

8.3.7 Invocation Expressions

An *invocation expression* is used to invoke a behavior, either directly by name or indirectly by calling an operation or sending a signal. An invocation expression consists of a *target,* which may be a behavior name, a behavioral feature reference or a `super` reference, and a *tuple,* which provides actual arguments for any parameters of the invoked behavior or behavioral feature.

Syntax

```
InvocationExpression(e: InvocationExpression)
      = InvocationTarget(e) Tuple(e.tuple)
InvocationTarget(e: InvocationExpression)
      = BehaviorInvocationTarget(e)
      | FeatureInvocationTarget(e)
      | SuperInvocationTarget(e)
```

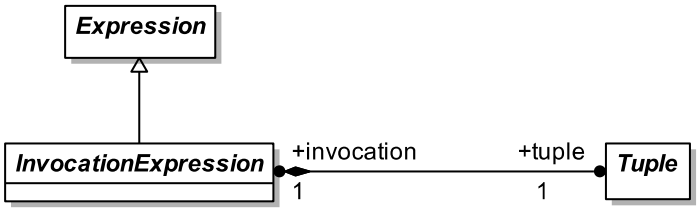

Figure 8.7 Abstract Syntax of Invocation Expressions

Cross References

1. Expression see 8.1
2. Tuple see 8.3.8
3. BehaviorInvocationTarget see 8.3.9
4. FeatureInvocationTarget see 8.3.10
5. SuperInvocationTarget see 8.3.11

Semantics

For each type of invocation expression, the target potentially specifies a set of parameters for which actual arguments need to be provided in the invocation. For `in` parameters, the argument is a value that is assigned to the corresponding parameter. For `inout` and `out` parameters, the argument must be an expression of the form that is legal on the *left hand side* of an assignment (see 8.8 on assignments).

An invocation expression may assign to names that are used as arguments for `out` or `inout` parameters. This is discussed as part of the semantics for tuples (see 8.3.8). The assigned source for a name after the invocation expression is the same as the assigned source after the tuple.

For a synchronous behavior or operation call, if the invoked behavior has a `return` parameter, then the values on that parameter at the completion of the invocation provide the result of the invocation expression. Otherwise the invocation expression produces no result values.

Specific semantics for each kind of invocation expression are further discussed in subclauses 8.3.9 to 8.3.11.

8.3.8 Tuples

A *tuple* is a list of expressions used to provide the arguments for an invocation. There are two kinds of tuples, *positional* tuples and *named* tuples. In a positional tuple, the arguments are matched to the parameters in order, by position. A named tuple, on the other hand, includes an explicit identification of the name of the parameter corresponding to each argument.

NOTE. The sequence operation expression notation is not available at the minimum conformance level (see 2.2).

Syntax

```
Tuple(t: Tuple)
      = PositionalTuple(t)
      | NamedTuple(t)
PositionalTuple(t: PositionalTuple)
      = "(" [ TupleExpressionList(t) ] ")"
```

```
TupleExpressionList(t: PositionalTuple)
      = Expression(t.expression) { "," Expression(t.expression) }
NamedTuple(t: NamedTuple)
      = "(" NamedExpression(t.namedExpression)
        { "," NamedExpression(t.namedExpression) } ")"
NamedExpression(n: NamedExpression)
      = Name(n.name) "=>" Expression(n.expression)
```

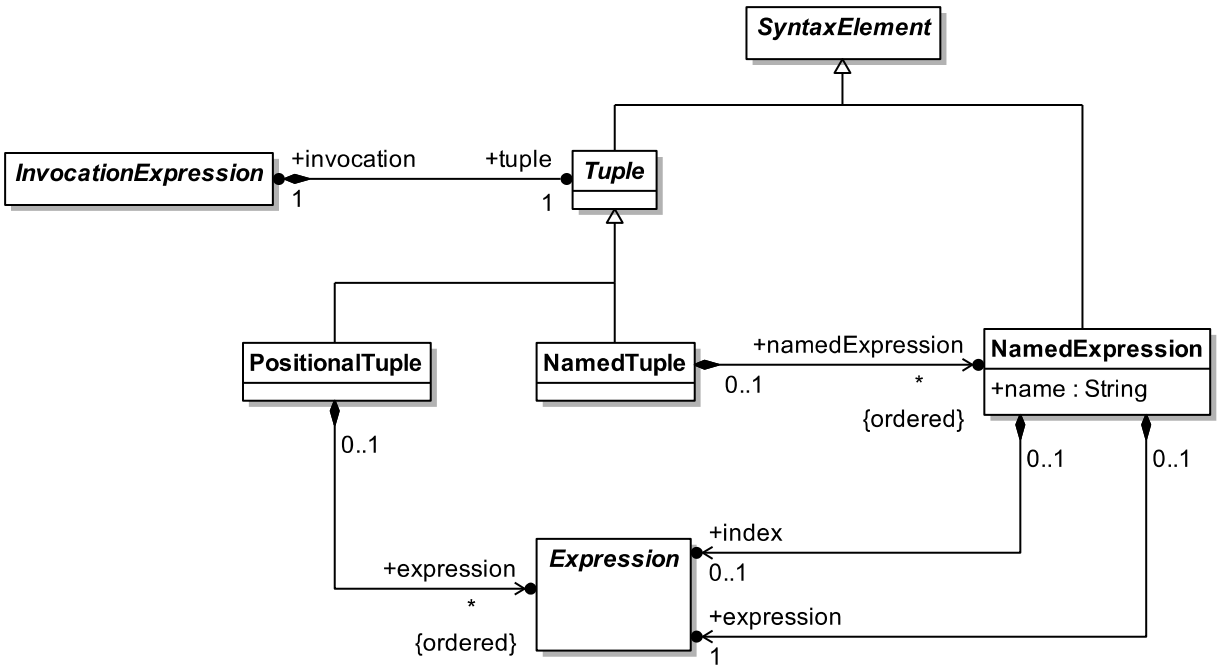

Figure 8.8 Abstract Syntax of Tuples

Cross References

1. Name see 7.6
2. Expression see 8.1
3. InvocationExpression see 8.3.7

Semantics

A tuple is evaluated by evaluating each of its constituent expressions *concurrently.* For arguments of `in` and `inout` parameters, the argument expression is fully evaluated to a value. For arguments of `out` parameters, the constituent parts of the argument expression are evaluated as if for the left hand side of an assignment (see 8.8).

NOTE. Since the argument expressions in a tuple are evaluated concurrently, they should not have side effects that influence each other's evaluation.

Arguments

In a positional tuple, each argument expression corresponds, in order, to a parameter for the invocation in which the tuple is used.

In a named tuple, the argument names must be parameter names for the invocation in which the tuple is used. The arguments may appear in any order, but a parameter name may appear at most once in the

tuple, and every non-optional parameter (i.e., having multiplicity lower bound greater than zero) of a target must be named in the invocation of that target.

A tuple may have fewer argument expressions than parameters. For a positional tuple, the unmatched parameters are those sequentially after the ones matched by the given argument expressions. For a named tuple, the unmatched parameters are those that are not named. An unmatched parameter must have mode `out` or have a multiplicity lower bound of 0.

For example, consider an activity with the following signature:

```
activity A(in x: Integer, in y: Boolean[0..1])
```

The following is then an invocation of this activity with a positional tuple:

```
A(1, true)
```

In this case, the argument `1` corresponds to the parameter `x` and the argument `true` corresponds to the parameter `y`. This is equivalent to the invocation with the named tuple:

```
A(x=>1, y=>true)
```

However, with a named tuple, the order of the arguments may be changed:

```
A(y=>true, x=>1)
```

Further, since the parameter `y` is optional (multiplicity lower bound of 0), no argument needs to be provided for it at all in the named tuple notation. For example, the following invocation using a named tuple:

```
A(x=>1)
```

is equivalent to the following invocation using a position tuple:

```
A(1)
```

which is in turn equivalent to:

```
A(1, null)
```

Assignment

Tuples with arguments corresponding to `inout` or `out` parameters act as assignments. The expressions corresponding to such parameters must have the form of the left hand side of an assignment (see 8.8), that is, either a local name or an attribute reference, possibly indexed. The assigned source for a name after the argument expression is determined as for the left hand side of an assignment, with the parameter as its assigned expression.

An argument expression for an `inout` parameter must also meet the static semantics of a name expression (see 8.3.3) or property access expression (see 8.3.6), if it is not index, or a sequence access expression (see 8.3.16), if it is indexed, but the argument expression for an `out` parameter does not. Thus, a local name used as the argument for an `out` parameter does not have to have an assigned source before the tuple, and a local name without a previous assigned source is considered to be newly defined with the same type and multiplicity as the `out` parameter.

A name may be assigned in at most one argument expression of a tuple. If there is one such argument expression, then the assigned source for the name after the tuple is the assigned source for the name after that argument expression. Otherwise the assigned source after the tuple is the same as before the tuple.

The result of an assignment to a name in one argument expression will not be used in other argument expressions. New local names defined in one argument expression cannot be used in another.

NOTE. This rule allows the argument expressions in a tuple to be evaluated concurrently.

8.3.9 Behavior Invocation Expressions

The simplest kind of invocation expression is the direct invocation of a behavior. In a *behavior invocation expression*, the target is given as the (qualified) name of the behavior to be invoked.

The same syntax may also be used with the qualified behavior of an *association end* as the target. In this case, the argument expressions in the tuple give the values for each of the other ends of the association. The result of the expression is a sequence of values of the target end for all links of the association that have the given values for the other ends.

Examples

Behavior Invocation

```
ComputeInterest(amount)
```

```
Start(monitor => systemMonitor)
```

```
including<Integer>(Integer[]{1,2,3}, 4)
```

Association Read

```
Roster::player(team=>t, season=>y)
```

```
Roster.player(t,y)
```

```
Owns::house(jack)
```

Syntax

```
BehaviorInvocationTarget(e: BehaviorInvocationExpression)
      = PotentiallyAmbiguousQualifiedName(e.target)
```

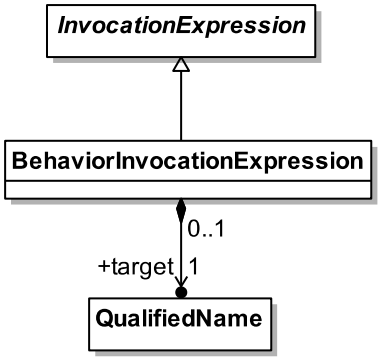

Figure 8.9 Abstract Syntax for Behavior Invocation Expressions

Cross References

1. QualifiedName see 8.2
2. PotentiallyAmbiguousQualifiedName see 8.2
3. InvocationExpression see 8.3.7

NOTE. See 8.2 for rules on the disambiguation of a qualified name with the dot notation initially parsed as a behavior invocation target to a feature invocation target.

Semantics

A behavior invocation expression is evaluated by first evaluating the argument tuple (see 8.3.8). The completion of the evaluation then depends on whether the target name resolves to a behavior or an association end. The given target name must identify either a visible behavior or an association end within the current scope of the behavior invocation expression. The target cannot be a template, though it may be a binding of a template behavior or association end (see 8.2).

Behavior Invocation

If the target is a behavior, then arguments are matched with parameters of the behavior as described in 8.3.8. Normally, each argument expression of the invocation must be *statically compatible* with the corresponding parameter. That is:

- For an `in` parameter, the argument expression must be *assignable* to the parameter (see 8.8 for the definition of assignability).
- For an `out` parameter, the parameter must be assignable to the argument expression.
- For an `inout` parameter, the argument expression and the parameter must be assignable to each other.

If the target behavior is not a template or it is the binding of template with arguments for all template parameters, then the above compatibility conditions can be checked directly based on the behavior signature. However, the target behavior is also allowed to be a template, as long as all unbound template parameters are classifier template parameters. In this case, the types to be used as arguments for the template parameters are *inferred* from the types of the argument expressions as follow:

- For each classifier template parameter, the argument type is the effective common ancestor of the types of all the argument expressions corresponding to the `in` and `inout` parameters of the template behavior that have the classifier template parameter as their type, if such an effective common ancestor exists (see 8.7 for the definition of effective common ancestor). If a relevant parameter has a multiplicity upper bound greater than 1 and the type of an argument expression is a collection class (see 11.7), then the type of the result of the `toSequence` operation of that collection class should be used, rather than the type of the argument expression itself.
- If there are no `in` or `inout` parameters that have a classifier template parameter as their type, or there is no effective common ancestor of the relevant argument expression types, then the argument type for that classifier template parameter is `any`.

The target of the behavior invocation is then considered to be an implicit binding of the behavior template, using the inferred argument types for each template parameter. If a classifier template parameter has constraining classifiers, the corresponding inferred argument must meet these constraints (see 8.2 on template binding with classifier constraints), or the invocation is illegal. If the implicit binding is legal, the static compatibility of the argument expressions for the invocation can then be checked in the normal way against the signature of this implicit binding. The invocation is illegal if the compatibility checks fail.

For example, the `including` function from the standard library `CollectionFunctions` package (see 11.6) has the signature

```
including<T>(in seq: T[] sequence, in element: T):
  T[] sequence
```

The invocation

```
including(Integer[]{1,2,3}, 4)
```

is then equivalent to

```
including<Integer>(Integer[]{1,2,3}, 4)
```

Since the literal `4` has type `Natural` (see 7.8.3) and the effective common ancestor of `Integer` and `Natural` is `Integer` (since `Integer` generalizes `Natural`—see 11.3.1). The result is a sequence of `Integer` values.

NOTE. The above rule for argument type inference does not attempt to account for behavior parameters that may have types that are template bindings using the behavior template parameters as arguments (for example, bindings of collection class with a behavior template parameter as an argument). This greatly simplifies the inference rule and is adequate in most cases. In particular, the fact that the standard collection functions in Alf can use the UML notion of multiplicity rather than collection classes in the typing of parameters reduces the need for a more complicated inference rule in cases involving sequences and collections.

When a behavior invocation expression is evaluated, the values resulting from the evaluation of each input argument expression are assigned to the appropriate `in` and `inout` parameter and the behavior is invoked. Once the behavior completes execution, the result values for each `inout` and `out` parameter are assigned to their corresponding output arguments. (Note that these assignments may involve implicit conversions, as discussed in 8.8.)

If the named behavior has a `return` parameter, then the behavior invocation expression evaluates to the value of that parameter. The type and multiplicity of the behavior invocation expression are the same as for the `return` parameter.

If the named behavior does not have a `return` parameter, then it evaluates to the null (empty) collection. It is untyped with multiplicity `[0..0]`.

Indexing

The primitive behaviors listed in Table 8.1, from the `SequenceFunctions` (see 11.4.6) and `CollectionFunctions` (see 11.6) library packages, all have `index` parameters used to indicate a position in an input sequence. Normally, indexing in Alf (and UML) is from a starting value of 1 for the first position. However, if a behavior invocation expression for one of the behaviors listed in Table 8.1 is contained, directly or indirectly, within a statement to which the annotation `@indexFrom0` applies (see 9.2), then the `index` argument is adjusted so that indexing is from 0.

Table 8.1 Library Primitive Behaviors with `index` Parameters

SequenceFunctions	CollectionFunctions
At	at
IncludeAt	includeAt
InsertAt	insertAt
IncludeAllAt	includeAllAt
ExcludeAt	excludeAt
ReplacingAt	replacingAt
	addAlt
	addAllAt
	removeAt
	replaceAt

For example, given the sequence

```
a = Integer[]{10, 20, 30, 40}
```

the behavior invocation expression `at(seq=>a, index=>3)` would normally evaluate to `30`, because that is the third element in the sequence. However, within the scope of an `@indexFrom0` annotation, the same expression would evaluate to `40`, because that is the element at index 3 when indexing starts at 0.

Further, within the scope of an `@indexFrom0` annotation, the value returned from invocations of the `SequenceFunctions::IndexOf` and `CollectionFunctions::indexOf` library functions is adjusted to return a 0-based rather than a 1-based index. For example, using the sequence from above, the behavior invocation expression `indexOf(a, 30)` would normally evaluate to `3`. However, within the scope of an `@indexFrom0` annotation, the same expression would evaluate to `2`.

Association Read

When an association end name is used in a behavior invocation expression, the association end may be thought of as a function from the values of the other association ends to the values on the target association. Arguments are matched to the ends of the association other than the target end. For the purposes of this matching, the non-target association ends are treated as if they were `in` parameters. Each argument expression must be statically compatible with the type of the corresponding association end, as described above.

The expression evaluates to all values of the target end of the links of the association whose other ends have the values given as arguments. The type and multiplicity of the invocation expression are the same as the type and multiplicity of the target association end.

For example, given the association

```
assoc Roster {
  public team: Team[*];
  public season: Year[*];
  public player: Player[*];
}
```

the expression

```
Roster::player(team=>t, season=>y)
```

or, equivalently,

```
Roster.player(t,y)
```

evaluates to the collection of players who played on team `t` in year `y`.

The behavior invocation expression notation for reading an association can be used for associations with any number of ends. This is in contrast to the property access expression notation (see 8.3.6), which can only be used to read binary associations. Of course, the behavior invocation form may also be used to read binary associations. So, for example, the expression

```
Owns::house(jack)
```

is equivalent to

```
jack.house
```

8.3.10 Feature Invocation Expressions

A *feature invocation expression* has a feature reference as its target. The referenced feature consists of a primary expression (see 8.3.1) and the name of either an operation or a reception owned by the type of the primary expression. If the named feature is an operation, then the feature invocation expression denotes a call to that operation. If the feature is a reception, then the feature invocation expression denotes sending an instance of the signal corresponding to the reception.

Examples

```
group.activate(nodes, edges)
```

```
actuator.Initialize(monitorRef => systemMonitor)
```

Syntax

```
FeatureInvocationTarget(e: FeatureInvocationExpression)
      = FeatureReference(e.target)
      | "this"
```

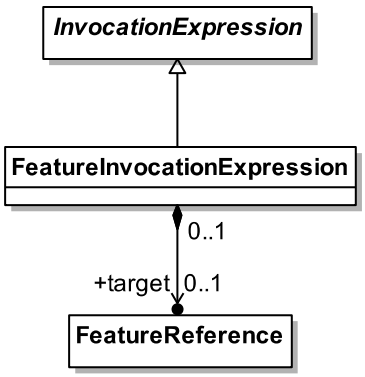

Figure 8.10 Abstract Syntax of Feature Invocation Expressions

Cross References

1. FeatureReference see 8.3.6
2. InvocationExpression see 8.3.7

NOTE. see 8.2 for rules on the disambiguation of a qualified name with the dot notation initially parsed as a behavior invocation target to a feature invocation target.

Semantics

If a feature invocation expression has a target feature reference, the primary expression in the feature reference must not be untyped. The named feature must denote either a visible operation or a visible reception of type of the primary expression, known as the *target type*. An operation call is distinguished from a signal send by whether the named feature is an operation or a reception. If the named feature is a template operation, then a template binding must be provided with arguments for all its template parameters. The primary expression in the feature reference is evaluated before the invocation argument expressions and names assigned in the primary expression are available in the argument expressions.

If the feature invocation target is the single keyword "`this`", the invocation expression must appear in the definition of the method of a constructor operation; that is, an operation with the standard stereotype «`Create`» (see UML Superstructure, 9.3.1). The name of the invoked operation is then implicitly taken to be the name of the owning classifier of the operation and the target type is taken to be that classifier.

Operation Call

If the named feature is an operation, then determining which operation to actually call is complicated by the possibility of operation overloading. The determination is made in the following steps:

1. Identify all concrete operations of the target type with the given name. If there is not at least one such operation, then the feature invocation expression is illegal.
2. From the set determined in Step 1, select the operations for which the tuple is statically compatible with the operation parameters, as defined for a behavior invocation (see 8.3.9). Due to the assignability rules, there may be more than one. If there is not at least one, then the feature invocation expression is illegal.
3. From the set determined in Step 2, select the most specific operations. One operation is *more specific* than another if it has fewer parameters or if it has the same number of parameters and
   - Each of its `in` parameters is assignable to the corresponding parameter of the other operation, in order (see 8.8 for the definition of assignability).
   - Each of the `out` or `return` parameters of the other operation is assignable to its corresponding parameter, in order.

   NOTE. Any corresponding `inout` parameters of operations remaining after Step 2 that have matching arguments will necessarily be assignable to each other.

   An operation in a set is *most specific* if there is no other operation in the set that is more specific than it.

If there are multiple operations remaining after the above steps, the feature invocation expression is illegal.

If there is a single operation remaining after the above steps, this is the operation to be called.

If the operation is a constructor, then the invocation expression is an *alternative constructor invocation.* Such an invocation is illegal unless it occurs in an expression statement (see 9.7) as the *first* statement in the definition for the method of a constructor operation. If the feature invocation target is the single keyword "`this`", then the identified operation with the same name as the target type *must* be a constructor, or the expression is illegal.

If there are no operations left after the above steps, the feature invocation expression is illegal, with one exception. If the operation name is "`destroy`" and the tuple is empty, then the feature invocation expression is legal and considered to be an *implicit object destruction expression.*

NOTE. The identifier "`destroy`" is *not* reserved and it is possible for a class to have an explicit parameterless operation called "`destroy`". By convention, any such operation should be a destructor, in which case it can be considered to be an explicit override of the implicit object destruction behavior.

If the feature invocation expression is for an operation with a `return` parameter, then the type of the feature invocation expression is the same as for the `return` parameter. For a single instance feature invocation, the multiplicity is also the same. For a collection feature invocation, the multiplicity is determined as for the corresponding `collect` expression (see 8.3.21), as described under Sequence Feature Invocation below.

If a feature invocation expression is for an operation without a `return` parameter, then it is untyped with multiplicity `[0..0]`.

Signal Send

If the named feature is a reception, then arguments are matched with attributes of the associated signal as described in 8.3.8, with the attributes being considered as parameters. Each argument expression must be assignable to the corresponding attributes (see 8.8 for the definition of assignability), the attributes being effectively considered as `in` parameters.

A feature invocation expression for a signal send is untyped with multiplicity `[0..0]`.

Single Instance Feature Invocation

A feature invocation expression is evaluated by first evaluating the primary expression, which results in a sequence of instances. The denoted invocation is then carried out on each element of the sequence.

If the primary expression of the feature reference for a feature invocation expression has multiplicity `[1..1]`, then the invocation expression is a *single instance feature invocation.* Otherwise it is a *sequence feature invocation.* An alternative constructor invocation must always be a single instance feature invocation.

For a single instance feature invocation, the result of the primary expression will always be a single instance. In this case, if the named feature is an operation, the invocation is a call to the named operation on the given instance. The tuple is evaluated to provide arguments for the operation parameters. The call is polymorphic, so, if the dynamic type of the instance has an operation that redefines the named operation (directly or indirectly), it is the redefined operation that is called.

If the operation is a destructor (i.e., it has the standard stereotype «`Destroy`»), then it is called just as a normal operation. However, after the completion of the call to the destructor, the target instance is destroyed, except if the target instance is the same as the current context object, in which case the destructor is called, but the instance is not destroyed.

If the expression is an implicit object destruction expression, then evaluation of the expression simply results in the target instance being destroyed, except, as above, if the target instance is the same as the current context object, in which case the object destruction expression has no effect.

If the named feature is a reception, the invocation is a sending of an instance of the signal associated with the reception. The tuple is evaluated to provide values for the attributes of the signal. Each signal attribute is treated as effectively an `in` parameter for the purposes of the invocation.

Sequence Feature Invocation

A sequence feature invocation expression of the form `primary.name tuple` is equivalent to a sequence expansion expression (see 8.3.21) of the form

```
primary -> collect x (x.name tuple)
```

NOTE. This means that the argument expressions in the tuple for the feature invocation are *re-evaluated* for the invocation on each instance in the sequence. It is not an error for the result of the primary expression in a feature invocation expression to be empty. In this case, no invocations occur and the tuple is never evaluated. It also means that, as would be the case for the equivalent sequence expansion expression, it is not allowed to reassign local names within the tuple of a sequence feature invocation (see also 8.3.19). A (parenthesized) null-coalescing expression (see 8.7) can be used as the primary of a feature invocation expression to ensure that it is a single instance feature invocation.

8.3.11 Super Invocation Expressions

A *super invocation expression* is used to invoke an operation of a superclass of the current context class. It is syntactically similar to a feature invocation expression (see 8.3.10), but with the keyword `super` used as the target. Unlike a feature invocation expression, however, a super invocation expression may name a superclass operation by a qualified name, if this is necessary in order to disambiguate operations with the same name from different superclasses.

Examples

```
super.run()
super.initialize(startValue)
super.Person::setName(name)
```

Syntax

```
SuperInvocationTarget(e: SuperInvocationExpression)
      = "super" [ "." QualifiedName(e.target) ]
```

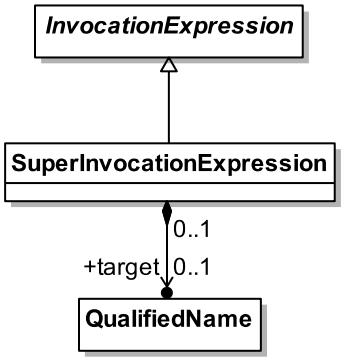

Figure 8.11 Abstract Syntax for Super Invocation Expressions

Cross References

1. QualifiedName see 8.2
2. InvocationExpression see 8.3.7

Semantics

If the super invocation target includes a qualified name with a qualification, then this qualification must resolve to one of the superclasses of the current context class, and the invoked operation must come from this superclass. If the given name is not qualified, then the invoked operation may come from *any* of the superclasses of the context class.

If the super invocation target is the single keyword "`super`" (with no qualified name), the invocation expression must appear in the definition of the method of a constructor operation; that is, an operation with the standard stereotype `«Create»` (see UML Superstructure, 9.3.1). The name of the invoked operation is then implicitly taken to be the name of the owning classifier of the operation, which must have a single superclass, from which the invoked operation is to come.

The operation to be called is determined using the following steps:

1. Identify all the concrete operations with the given name that are members of the relevant superclass or superclasses (as discussed above). If there is not at least one such operation, then the super invocation expression is illegal.
2. From the set determined in Step 1, select the operations for which the tuple is statically compatible with the operation parameters, as defined for a behavior invocation (see 8.3.9). Due to the assignability rules, there may be more than one. If there is not at least one, then the super invocation expression is illegal.
3. From the set determined in Step 2, select the most specific operations, as defined in Step 3 of the determination of the operation for a feature invocation expression (see 8.3.10). If there is a single operation remaining, this is the operation to be called. Otherwise, the super invocation expression is illegal.

If the identified operation is a constructor, then the invocation expression is a *super constructor invocation.* Such an invocation is illegal unless it occurs in an expression statement (see 9.7) at the start of the definition for the method of a constructor operation such that any statements preceding it are also super constructor invocations. If the super invocation target is the single keyword "`super`", then the identified operation *must* be a constructor, or the expression is illegal.

If the identified operation is a destructor (i.e., it has the standard stereotype `«Destroy»`), then the super invocation expression must itself appear within the method of a destructor operation.

When a super invocation expression is evaluated, its tuple is first evaluated to provide arguments for the operation parameters. The method of the named superclass operation is then called on the current context object. Note that the call is *not* polymorphic—the statically determined superclass method behavior is always directly invoked.

If the super invocation expression has a `return` parameter, then the type and multiplicity of the super invocation expression is the same as for the `return` parameter. If the operation does not have a `return` parameter, then the super invocation expression is untyped with multiplicity `[0..0]`.

8.3.12 Instance Creation Expressions

An *instance creation expression* is used to create a new instance of a class or data type. In either case, an instance creation expression consists of the keyword `new` followed by a (possibly qualified) name and a tuple.

Examples

Object Creation

```
new Employee(id, name)
```

```
new Employee::transferred(employeeInfo)
```

```
new Set<Integer>(Integer[]{1,2,3})
```

Data Value Creation

```
new Position(1,2)
```

```
new Position(x=>1, y=>2)
```

Syntax

```
InstanceCreationExpression(e: InstanceCreationExpression)
      = "new" QualifiedName(e.constructor) Tuple(e.tuple)
```

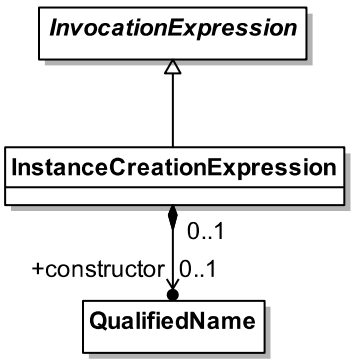

Figure 8.12 Abstract Syntax of Instance Creation Expressions

Cross References

1. QualifiedName see 8.2
2. InvocationExpression see 8.3.7
3. Tuple see 8.3.8

Semantics

Since an instance creation expression involves the invocation of a constructor operation, it is possible for it to assign to names used as arguments for `out` or `inout` parameters, as for a regular invocation expression (see 8.3.7). The name given in an instance creation expression must resolve to a class, data type or a constructor operation. If the name given in an instance creation expression denotes both a class and a data type, then the expression is illegal. If the qualified name denotes a data type, but not a class, then the instance creation expression is a data value creation expression. Otherwise it is an object creation expression.

The element named in an instance creation expression may not be a template, though it may be a binding of a template with arguments given for all template parameters.

Object Creation Expression

If the name in an instance creation expression denotes a constructor operation or a class, then the expression creates an object. If the name is for a constructor, then the newly created object is an instance of the class that owns the constructor. Otherwise, the object is an instance of the explicitly named class.

If the name denotes neither a class nor a data type, then it must denote a constructor (see UML Superstructure, 9.3.1), which is determined using the following steps:

1. Identify all the constructors with the given name. Due to overloading there may be more than one. If there is not at least one, then the object creation expression is illegal.
2. From the set determined in Step 1, select the constructors for which the tuple is statically compatible with the constructor parameters, as defined for a behavior invocation (see 8.3.9). Due to the assignability rules, there may be more than one. If there is not at least one, then the object creation expression is illegal.
3. From the set determined in Step 2, select the most specific constructors, as defined in Step 3 of the determination of the operation for a feature invocation expression (see 8.3.10). If there is a single constructor remaining, this is the constructor to be called. Otherwise, the object creation expression is illegal.

NOTE. A constructor is always required to have a single return type that is the same as the class being constructed (see UML Superstructure, subclauses 9.3.1).

If the name denotes a class, then the constructor to be used is determined as above, except that, in the first step, constructors are identified that are owned members of the named class with the same name as the class. Further, if no constructor is found, then the object creation expression is considered to be *constructorless*. However, a constructorless instance create expression may not have any arguments and is only legal if the named class has no constructor operations (see 10.5.3.1).

The class of the object being created must *not* be abstract, unless all of the following conditions hold:

- The object creation expression is not constructorless.
- The namespace that owns the class of the constructor also owns a package with the name `Impl`.
- The `Impl` package contains a class with the same name as the class of the constructor.
- The `Impl` class has a constructor that redefines the constructor (which implies that the Impl class must be a direct or indirect subclass of the class of the constructor).

If these conditions hold, then the identified `Impl` class constructor is used instead of the original abstract class constructor, and the object that is created is actually an instance of the `Impl` class.

NOTE. The above mechanism is intended to allow for the definition of abstract classes in model libraries that may be directly referenced by user models and constructed as if they were concrete. Different execution tools may provide different actual concrete implementations of the library classes in the nested `Impl` package of the model library without changing the library classes actually referenced in user models. In particular, the Alf standard model library `CollectionClass` package uses this mechanism (see 11.7).

If an object creation expression is *not* constructorless, then, in addition to creating an object, evaluation of the expression calls the identified constructor on the newly created object. If a constructor is explicitly named in the expression, then that is the constructor that is called. For example, the expression

```
new Employee::transferred(employeeInfo)
```

creates an object of the class `Employee` and calls the constructor `transferred` on that object with the argument `employeeInfo`.

If the object creation expression names a class, then the constructor called is one with the same name as the class. Thus, the expression

```
new Employee(id, name)
```

is equivalent to

```
new Employee::Employee(id, name)
```

If an object creation expression is constructorless, then evaluation of the expression still results in the creation of a new object of the named class, but no constructor operation is called. However, if any attributes of the class have default value expressions (see UML Superstructure, 7.3.44), then these are evaluated to give the initial values of the corresponding attributes. Such initialization has the semantics of an assignment of the expression to the attribute (see 8.8). Attributes are initialized in the order in which they are defined in the class.

Finally, if the class of the object being created is active, then the classifier behavior for that class is automatically started once the object is created and after any constructor call completes. In addition, each classifier behavior of a direct or indirect active superclass is also started, unless that classifier behavior is redefined in another superclass or the class of the object being created.

NOTE. While UML allows for the redefinition of behaviors (see UML Superstructure, 13.3.2), this is not included in the fUML subset (see fUML Specification, 7.3.2.1). Nevertheless, Alf notation may be used to create an instance of a class with a classifier behavior that redefines the classifier behavior of its superclass. For example, such redefinition is common when state machines are used as classifier behaviors. The intent is that the behavior specified in the redefined behavior is automatically included in the redefining behavior, which only specifies incremental and compatible extensions to the redefined behavior. Therefore, it would be redundant in this case to start both the redefined and redefining behaviors. Only the redefining behavior should be started.

The type of an object creation expression is the class that owns the constructor (which is the named class, if it is named explicitly) and the multiplicity is `[1..1]`.

Data Value Creation Expression

If the name in an instance creation expression denotes a data type, then the expression creates a data value. In this case, the tuple is used to specify values for the attributes of the data value. If a named tuple is used, then the names must correspond to the names of the attributes of the data type. The identified data type must not be abstract.

Arguments are matched with attributes of the named data type as described in 8.3.8, with the attributes being considered as `in` parameters. Each argument expression must be assignable to the corresponding attribute (see 8.8 for the definition of assignability).

For example, consider the data type

```
datatype Position {
  public x: Integer;
  public y: Integer;
}
```

All the following data value expressions create equivalent data values of this type:

```
new Position(1,2)

new Position(x=>1, y=>2)

new Position(y=>2, x=>1)
```

The type of a data value creation expression is the named data value and the multiplicity is `[1..1]`.

8.3.13 Link Operation Expressions

A *link operation expression* is used to create or destroy instances of a named association, known as *links*. A link operation expression has a similar syntax to an invocation expression, consisting of a *target* association name, a link operation name and a tuple of actual arguments for the link operation.

Examples

```
Owns.createLink(jack, newHouse)

Owns.createLink(owner=>jack, house=>newHouse)

Owns.createLink(owner=>jack, house[1]=>newHouse)

Owns.destroyLink(owner=>jack, house=>newHouse)

Owns.clearAssoc(jack)
```

Syntax

```
LinkOperationExpression(e: LinkOperationExpression)
      = QualifiedName(e.associationName) "." LinkOperation(e.operation)
        LinkOperationTuple(e.tuple)
LinkOperation(op: String)
      = "createLink"(op)
      | "destroyLink"(op)
      | "clearAssoc"(op)
LinkOperationTuple(t: Tuple)
      = PositionalTuple(t)
      | IndexedNamedTuple(t)
IndexedNamedTuple(t: NamedTuple)
      = "(" IndexedNamedExpression(t.expressions)
        { "," IndexedNamedExpression(t.expressions) } ")"
IndexedNamedExpression(n: NamedExpression)
      = Name(n.name) [ Index(n.index) ] "=>" Expression(n.expression)
Index(e: Expression)
      = "[" Expression(e) "]"
```

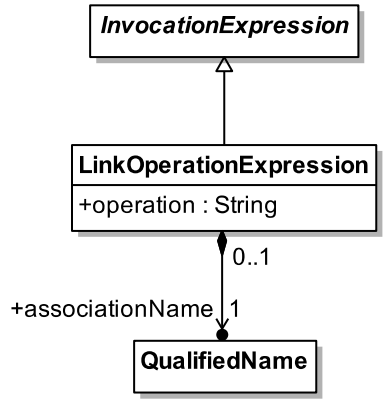

Figure 8.13 Abstract Syntax of Link Operation Expressions

Cross References

1. Name see 7.6

2. Expression see 8.1
3. QualifiedName see 8.2
4. InvocationExpression see 8.3.7
5. PositionalTuple see 8.3.8
6. NamedTuple see 8.3.8
7. NamedExpression see 8.3.8

Semantics

The target of a link operation expression is a qualified name that must resolve to an association. The expression must name one of the link operations in Table 8.2 (note that link operation names are reserved words). These operations are used to create or destroy links of the named association.

Table 8.2 Link Operations

Operation	Description
`A.createLink(e1,e2…)`	Create a link of association `A` with end values `e1`, `e2`, … Association `A` must not be abstract.
`A.destroyLink(e1,e2,…)`	Destroy a link of association `A` with end values `e1`, `e2`, …
`A.clearAssoc(e)`	Destroy all links of association `A` with at least one end value `e`.

Argument expressions in the tuples for a link operation other than `clearAssoc` are matched to the association ends of the named association, where the ends are treated as if they were `in` parameters (see 8.3.8). Indexes are only allowed on the names for ordered association ends. The expression in an index must have a type that conforms to the type `UnlimitedNatural` and a multiplicity upper bound of 1.

The link operation `clearAssoc` must have a positional tuple with a single argument.

A link operation expression is untyped with multiplicity `[0..0]`.

For example, given the association

```
assoc Owns {
  owner: Person;
  house: House[*];
}
```

the expression

```
Owns.createLink(jack, newHouse)
```

creates a link with the given end values (note that the order of the arguments corresponds to the order of the association ends in the association definition). This link can then be destroyed using the expression

```
Owns.destroyLink(jack, newHouse)
```

Named tuple notation may also be used:

```
Owns.createLink(owner=>jack, house=>newHouse)
```

and

```
Owns.destroyLink(owner=>jack, house=>newHouse)
```

in which case the order of the arguments is immaterial.

If an association end is ordered, then the position of a link for the end can be indicated using an index. For example, if the association above was modified so that the `house` end is ordered, then the expression

```
Owns.createLink(owner=>jack, house[1]=>newHouse)
```

inserts the `newHouse` at the beginning of the list of houses for `jack`. If an index is not given for an ordered end, then the default is `*`, which indicates adding at the end.

Normally, indexing in Alf (as in UML) is from 1. However, if a link operation expression is contained, directly or indirectly, within a statement to which the annotation `@indexFrom0` applies (see 9.2), then indexing is from 0. Thus, if the example given above where in the scope of an `@indexFrom0` annotation, then `newHouse` would be inserted at the *second* position in the list of houses for `jack`, rather than at the beginning, because an index of `1` indicates the second position when indexing starts at 0.

Finally, there is an additional link operation, `clearAssoc`, which may only be used with associations. It destroys all links of the named association that have at least one end with a given value. Thus, the expression

```
Owns.clearAssoc(jack)
```

destroys all links between jack and any house.

NOTE. For a binary association (such as the example `Owns` used above), links may also be effectively created and destroyed using property access notation, as if association ends were properties of their opposite types. Thus the expression `add(jack.house,newHouse)` (or `jack.house->add(newHouse)`) can be used to create an `Owns` link, `remove(jack.house, newHouse)` (or `jack.house->remove(newHouse)`) to destroy it and `jack.house = null` to clear the association. (see 8.3.6 on property access expressions and 8.3.6 on sequence operation expressions.)

8.3.14 Class Extent Expressions

A *class extent expression* returns a sequence of the objects in the extent of a named class.

Examples

```
Customers.allInstances()
```

Syntax

```
ClassExtentExpression(e: ClassExtentExpression)
      = QualifiedName(e.type) "." "allInstances" "(" ")"
```

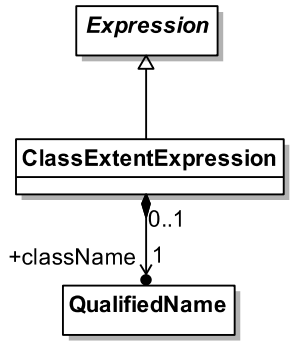

Figure 8.14 Abstract Syntax for Class Extent Expressions

Cross References

1. Expression see 8.1
2. QualifiedName see 8.2

Semantics

The name given in a class extent expression must denote a class. A class extent expression evaluates to a sequence (in an arbitrary order) of the objects in the extent of the named class. The *extent* of a class is the set of objects that currently exist at the specific execution locus at which the class extent expression is evaluated.

NOTE. The concept of an execution *locus* is defined in the fUML Specification, 8.2. The correspondence of the actual execution environment to one or more fUML loci is specific to the execution tool.

The type of a class extent expression is the named class, and its multiplicity is `[0..*]`.

8.3.15 Sequence Construction Expressions

A *sequence construction expression* is used to group values into a sequence of a specific type. The most direct form of a sequence construction expression is a list of expressions enclosed in braces and preceded by a specification of the desired type with the multiplicity indicator "`[]`". There is also a special notation for the case of a sequence of consecutive integers.

A sequence construction expression may also be used to create an instance of a collection class (see 11.7) initialized from the given sequence of values. This form has the same syntax as above, except that the specified type must be a collection class and no multiplicity indicator is included.

A sequence construction expression may optionally start with the keyword `new`, analogously to the syntax of an instance creation expression (see 8.3.12).

Examples

```
Integer[]{1, 3, 45, 2, 3}

Set<Integer>{1, 3, 45, 2, 3}

new String[]{"apple","orange","strawberry",}

new List< List<String> >{{"apple","orange"},{"strawberry","raspberry"}}

Integer[]{1..6+4}

null
```

Syntax

```
SequenceConstructionExpression(e: SequenceConstructionExpression)
      = NullExpression (e.hasMultiplicity=true)
      | SequenceElementsExpression(e)
NullExpression
      = "null"
SequenceElementsExpression(e: SequenceConstructionExpression)
      = [ "new" ] SequenceElementsTypePart(e)
        "{" SequenceElements(e.elements) "}"
SequenceElementsTypePart(e: SequenceConstructionExpression)
      = TypeName(e.typeName)
        [ MultiplicityIndicator (e.hasMultiplicity=true) ]
```

```
MultiplicityIndicator
      = "[" "]"
SequenceElements(se: SequenceElements)
      = SequenceElementList(se)
      | SequenceRange(se)
SequenceElementList(sel: SequenceExpressionList)
      = [ SequenceElement(sel.element) { "," SequenceElement(sel.element) }
        [ ","] ]
SequenceElement(e: Expression)
      = Expression(e)
      | SequenceInitializationExpression(e)
SequenceInitializationExpression(e: SequenceConstructionExpression)
      = [ "new" ] "{" SequenceElements(e.elements) "}"
SequenceRange(sr: SequenceRange)
      = Expression(sr.rangeLower) ".." Expression(sr.rangeUpper)
```

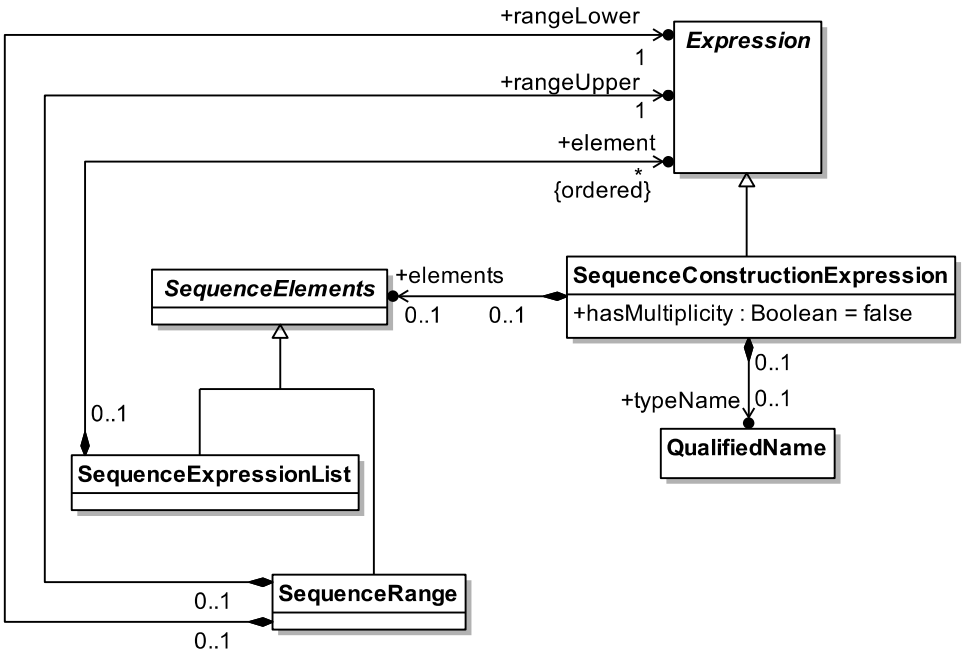

Figure 8.15 Abstract Syntax of Sequence Construction Expressions

Cross References

1. Expression see 8.1
2. TypeName see 8.2
3. ExpressionList see 8.3.8

Semantics

Type Part

A sequence construction expression begins with a *type part* that that consists of a type name (see 8.2) and an optional *multiplicity indicator* "`[]`".

If a multiplicity indicator is included, the type name may be either a qualified name or the keyword `any`. If it is a qualified name, then this name must resolve to a classifier, which is the type of the sequence

construction expression. The qualified name must not resolve to a template, though it may be the binding of a template classifier.

If a multiplicity indicator is not included, then the type name must be the qualified name of a collection class (see 11.7). In this case, the sequence construction expression has the collection class as its type.

Sequence Elements

The type part of a sequence construction expression is followed by a specification of the elements of a sequence. This may be given either as a sequence element list or a sequence range.

A *sequence element list* is an explicit list of expressions. Each expression in the list must have a multiplicity upper bound of no more than 1. The multiplicity lower and upper bounds of the sequence element list are given by the sum of the multiplicity lower and upper bounds of each of the expressions in the list. A sequence element list is evaluated by evaluating each expression in the list, in order, each of which will return at most one value. The result of the sequence element list is the sequence of values returned, in order.

A *sequence range* has the form `Expr1..Expr2`, where both expressions are of a type that conforms to type `Integer` and have a multiplicity upper bound of 1. A sequence range denotes all integers from the value of the first expression up to and including the value of the second expression. Note that the two expressions in a sequence range are evaluated *concurrently.* For example, the sequence range `{1..6+4}` is equivalent to the sequence element list `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`. The multiplicity of the sequence range is `[0..*]`.

If the type part of the sequence construction expression has a multiplicity indicator, then the result of the expression is the result of evaluating the sequence element specification for the expression. If the expression has a sequence element list, and the type name in the type part is not `any`, then the type of each expression in the list must either conform to the given type or be convertible to it by bit string conversion or real conversion (see 8.8 on type conformance, bit string conversion and real conversion).

Collection Object Creation

If the type part of the sequence construction expression does *not* have a multiplicity indicator, then the expression is equivalent to an instance creation expression (see 8.3.12) with a constructor for the given collection class and the sequence elements used as the constructor argument. The result of the expression is a single collection object, so its multiplicity is `[1..1]`.

For example, the sequence construction expression

```
Set<Integer>{1, 2, 3, 4}
```

(where `Set` is a collection class from the Alf Standard Model Library, see 11.7) is equivalent to the instance creation expression

```
new Set<Integer>(Integer[]{1, 2, 3, 4})
```

When used to construct a collection object, it is possible that the elements of a sequence element list are themselves collections. In this case, the type parts of the nested sequence construction expressions may be omitted, since their types may be inferred from the element type of the collection class being constructed.

For example, the sequence construction expression

```
List< Set<Integer> >{{1}, {2,3}, {4,5,6}}
```

is equivalent to

```
List< Set<Integer> >{Set<Integer>{1}, Set<Integer>{2,3}, Set<Integer>{4,5,6}}
```

which is in turn equivalent to the following nested instance creation expression:

```
new List< Set<Integer> >(Set<Integer>[]{
  new Set<Integer>(1),
  new Set<Integer>(Integer[]{2,3}),
  new Set<Integer>(Integer[]{4,5,6})
})
```

Note that this nesting of sequence construction expressions only applies to the case of the construction of a collection object. Sequences of values are themselves *flat* in Alf. Thus, the following expression is not legal:

```
Integer[]{{1}, {2,3}, {4,5,6}}  // Illegal!
```

Empty Sequences

The list of element expressions in a sequence construction expression may be empty. However, the static type of an empty sequence is still as specified in the expression. Thus, `Integer[]{}` is an empty sequence of integers while `String[]{}` is an empty sequence of strings. An empty sequence has multiplicity `[0..0]`.

The keyword `null` is equivalent to `any[]{}`, that is, an untyped empty sequence.

8.3.16 Sequence Access Expressions

A *sequence access expression* is used to obtain a specific element of an expression. It contains two subexpressions, a primary expression and an *index expression* (within brackets).

Examples

```
this.getTypes()[1]
```

Syntax

```
SequenceAccessExpression(e: SequenceAccessExpression)
      = PrimaryExpression(e.primary) Index(e.index)
```

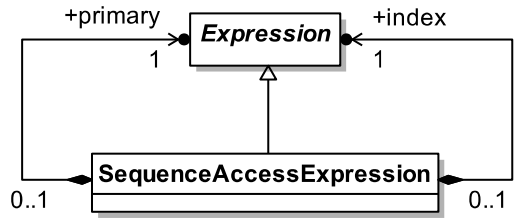

Figure 8.16 Abstract Syntax of Sequence Access Expressions

Cross References

1. Expression see 8.1
2. PrimaryExpression see 8.3.1
3. Index see 8.3.13

Semantics

The two subexpressions of a sequence access expression are evaluated concurrently. The index expression must evaluate to a single integer value that indicates the element of the collection to which the collection access expression evaluates.

Indexing is from 1, unless the sequence access expression is contained, directly or indirectly, within a statement to which the annotation `@indexFrom0` applies (see 9.2), in which case indexing is from 0. If the index value is less than 1 (0, if indexing is from 0) or greater than the size of the collection (size of the collection minus 1, if indexing is from 0), then the expression returns no value.

For example, given the sequence

```
a = Integer[]{10, 20, 30, 40}
```

the sequence access expression `a[3]` normally evaluates to `30`. However, if the expression is within the scope of an `@indexFrom0` annotation, then it evaluates to `40`, because that element is at index `3` when indexing starts at 0.

If the index value is less than 1 or greater than the size of the collection, then the expression returns no value.

The type of a sequence access expression is the same as the type of its primary expression. Its multiplicity is `[0..1]`.

8.3.17 Sequence Operation Expressions

A *sequence operation expression* is an alternative notation for applying a behavior as an operation on a sequence of values. When the target of a normal operation invocation expression evaluates to a sequence of values, the operation is invoked on each object in the sequence (see 8.3.10). A sequence operation expression, on the other hand, can be used to invoke in an operation-like way a behavior that is intended to operate on a sequence *as a whole.*

Similarly to an operation invocation expression, a sequence operation expression consists of a primary expression, an operation name and a tuple. However, the primary expression and the operation name are separated by the symbol "`->`" rather than the "`.`" used for a normal operation invocation. (The symbol "`->`" can be thought of as indicating the "flow" of a sequence of values into the sequence operation.)

NOTE. The sequence operation expression notation is not available at the minimum conformance level (see 2.2).

Examples

```
selectedCustomers->notEmpty()
```

```
memberList->includes(possibleMember)
```

```
memberList->including(newMember)
```

```
products->removeAll(rejects)
```

Syntax

```
SequenceOperationExpression(e: SequenceOperationExpression)
      = ExtentOrExpression(e.primary) "->" QualifiedName(e.operation)
        Tuple(e.tuple)
ExtentOrExpression(e: ExtentOrExpression)
      = QualifiedName(e.name)
      | NonNamePrimaryExpression(e.nonNameExpression)
```

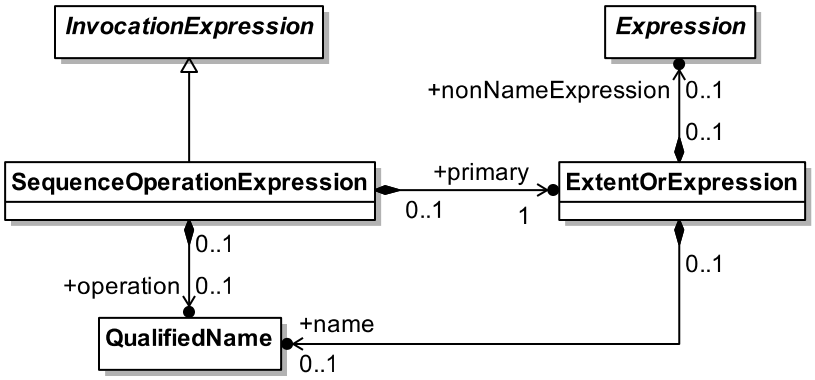

Figure 8.17 Abstract Syntax for Sequence Operation Expressions

Cross References

1.	Expression	see 8.1
2.	QualifiedName	see 8.2
3.	NonNamePrimaryExpression	see 8.3.1
4.	InvocationExpression	see 8.3.7
5.	Tuple	see 8.3.8

Semantics

Sequence Operation Invocation

The "operation" name in a sequence operation expression does not actually name an operation. Instead, it must resolve to the name of a behavior with at least one parameter. The first parameter of this behavior must have direction `in` or `inout` and a multiplicity of `[0..*]`. The primary expression of the link operation expression must be assignable to this parameter (see 8.8 for the definition of assignability).

The tuple of a sequence operation expression contains arguments for any parameters of the named behavior other than the first. Argument matching and compatibility is as required for a behavior invocation (see 8.3.9).

While a sequence operation expression is intended to look notationally similar to an operation invocation, it is actually executed equivalently to a behavior invocation expression (see 8.3.9). Specifically, the target primary expression and any argument expressions in the tuple are all evaluated *concurrently.* The named behavior is then invoked on the resulting arguments as for a behavior invocation.

For example, suppose `a` is a sequence. Then the sequence operation expression

```
a->including(x)
```

is equivalent to the following behavior invocation expression for the library behavior `including`:

```
including(a,x)
```

Both of these expression evaluate to a sequence with all the elements of `a` and the additional element `x` at the end.

By contrast, suppose that `a` is a sequence of objects of the same class, which has an operation called `including`. Then, the operation call

```
a.including(x)
```

results in `including(x)` being called on *each* member of the sequence `a`.

If the sequence operation expression is contained, directly or indirectly, within a statement to which the annotation `@indexFrom0` applies (see 9.2), then the invocation of certain library functions involving indexing is adjusted exactly as for the equivalent behavior invocation expression (as described in 8.3.9).

"In-Place" Sequence Operations

If the first parameter of the behavior has direction `inout`, then the primary expression for the sequence operation expression is restricted to the syntax and static semantic constraints of the left-hand side of an assignment (see 8.8). If the primary expression is a local name or parameter name, then the assigned source for that name after the sequence operation expression is the sequence operation expression itself. If the primary expression is an attribute reference with a local name or parameter name and has a type that is a structured data type, then the assigned source for that name after the sequence operation expression is the sequence operation expression itself.

The Alf standard `CollectionFunctions` package (see 11.6) contains a number of behaviors that operate on sequences. The majority of these have an `in` direction for their target sequence parameter and produce a result via a return parameter. However, certain of them have an `inout` direction for their sequence parameter, with the effect of updating a sequence "in place".

For example,

```
a->add(2)
```

is equivalent to

```
add(a,2)
```

which results in the same effective behavior as

```
a = a->including(2)
```

Clearly, in this case, the target primary expression must have the form of the left hand side of an assignment (see 8.8). Note that if the target primary expression is a local name (such as "`a`" in the example above), then, by the usual semantics of `inout` parameters, such an "in place" sequence operation expression is considered to be a re-assignment of that local name (see 8.3.8 on assignments and `inout` parameters).

Sequence Operations on Collections

The assignability rule for collection conversion (see 8.8) leads to special behavior in the case in which the type of the initial primary expression is a collection class (see 11.7) and its multiplicity upper bound is no greater than 1 (and as long as the initial parameter is an `in` parameter). In this case, the operation `toSequence` is first called on the collection object resulting from evaluating the primary expression, and the sequence operation is then performed on the resulting sequence.

For example, suppose `customerList` has the type `List<Customer>`. Then the expression

```
customerList->size()
```

is equivalent to

```
size(customerList)
```

Since the argument type for the `CollectionFunctions::size` has multiplicity `[*]`, this is equivalent to

```
size(customerList.toSequence())
```

which gives the number of elements in the collection. Note that this gives the same result as a regular call on the `size` operation provided by a collection object:

```
customerList.size()
```

Sequence Operations on Extents

There is also a special notation option for sequence operation expressions in which the initial primary expression may be replaced with the name of a class to implicitly denote the collection that is the current extent of that class. If the primary expression of a sequence operation expression is a class name, then it is considered to be equivalent to a class extent expression for the named class (see 8.3.14). However, if there is a local name or parameter name in the current extent with the same name as the class name, then the primary expression is considered a name expression for the given name (see 8.3.3) rather than a class extent expression.

For example, the expression

```
Customer->size()
```

is equivalent to

```
Customer.allInstances()->size()
```

or

```
size(Customer.allInstances())
```

which all evaluate to the number of objects currently in the extent of the `Customer` class (where `Customer.allInstances()` is a class extent expression—see 8.3.14).

8.3.18 Sequence Reduction Expressions

A *sequence reduction expression* is used to reduce a sequence of values to a single value by combining the elements of the sequence using a binary operator. A sequence reduction expression has a similar syntax to a sequence operation expression (see 8.3.17) in that it starts with a primary expression followed by the "flow" symbol "`->`". This is then followed by the keyword "`reduce`" and the qualified name of a behavior that acts as the binary operator over elements of the sequence.

As for sequence operation expressions (see 8.3.17), sequence reduction expressions also allow the special notation in which a class name is used as a shorthand for a class extent expression.

NOTE. The sequence reduction expression notation is not available at the minimum conformance level (see 2.2).

Examples

```
subtotals->reduce '+'
```

```
rotationMatrices->reduce ordered MatMult
```

Syntax

```
ReductionExpression(e: ReductionExpression)
      = ExtentOrExpression(e.primary) "->" "reduce"
        [ "ordered" (e.isOrdered=true) ] QualifiedName(e.behavior)
```

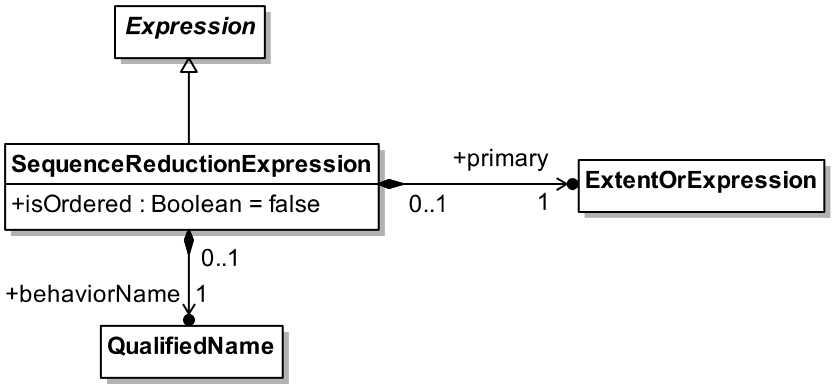

Figure 8.18 Abstract Syntax for Sequence Reduction Expressions

Cross References

1. Expression see 8.1
2. QualifiedName see 8.2
3. ExtentOrExpression see 8.3.17

Semantics

The qualified name in a sequence reduction expression must denote a behavior with two `in` parameters and a `return` parameter and no other parameters. The parameters must all have the same type as the primary expression and multiplicity `[1..1]`. The identified behavior must not be a template, though it may be a binding of a template behavior (see 8.2)

The named behavior is invoked repeatedly on pairs of values resulting from the evaluation of a target primary expression. Each time it is invoked, it produces one output that replaces the two input elements in an intermediate version of the sequence. This repeats until the sequence is reduced to a single value.

For example, the expression

```
Integer[]{1, 2, 3} -> reduce '+'
```

evaluates to 6—the same as `1+2+3` (see also 8.3.15 on the sequence construction expression notation used in the primary expression above).

Normally, the order in which the behavior of a sequence reduction expression is applied to pairs of values is indeterminate. This will not affect the result of the expression if the behavior is commutative and associative. For example, in the above example using addition may be evaluated as `1+2+3` or `1+3+2` or `2+3+1`, or in any other order, and the result is always 6.

However, if the reducing behavior is not commutative and associative, or has side effects, then the order in which elements are selected from the sequence will affect the result of the expression. In this case, an *ordered reduction* may be used by adding the keyword "`ordered`", so the reducing behavior will be applied to adjacent pairs according to the collection order. The result of each invocation of the behavior replaces the two values taken as input at the same position in the order as the original two values.

For example, matrix multiplication is not commutative. The expression

```
Matrix[]{A, B, C} -> reduce ordered MatMult
```

will be deterministically be evaluated as `MatMult(MatMult(A,B),C)`, not in any other order.

A sequence reduction expression has the same type as its argument expression and multiplicity `[1..1]`.

Even though a sequence reduction expression is not equivalent to a behavior invocation, a collection object may be directly acted on in such an expression analogously to how it may be acted on in a sequence operation expression (see 8.3.17). The operation `toSequence` is called on the collection object, and the reduction is applied to the result of that operation.

For example, if `vector` has the type `List<Integer>`, then the sequence reduction expression

```
vector -> reduce '+'
```

is equivalent to

```
vector.toSequence() -> reduce '+'
```

Also as for sequence operation expressions (see 8.3.17), sequence reduction expressions allow the special notation in which a class name is used as a shorthand for a class extent expression.

8.3.19 Sequence Expansion Expressions

A *sequence expansion expression* is used to specify a computation that operates on all the elements of a sequence. Such an operation is said to *expand* the collection. In all cases except the `iterate` operation, this expansion is *parallel,* in the sense that the computations on each sequence element are carried out concurrently.

A sequence expansion expression has a similar syntax to a sequence operation expression (see 8.3.17), except that, in addition to giving the name of the expansion operation, the expression also gives a name for an *expansion variable.* This expansion variable is used to hold the value of each element in the sequence during the expansion computation.

NOTE. The sequence expansion expression notation is not available at the minimum conformance level (see 2.2).

Syntax

```
SequenceExpansionExpression(e: SequenceExpansionExpression)
      = ExtentOrExpression(e.primary) "->" ExpansionOperation(e)
        Name(e.variable) "(" Expression(e.argument) ")"
ExpansionOperation(e: SequenceExpansionExpression)
      = SelectOrRejectOperation(e)
      | CollectOrIterateOperation(e)
      | ForAllOrExistsOrOneOperation(e)
      | IsUniqueOperation(e)
```

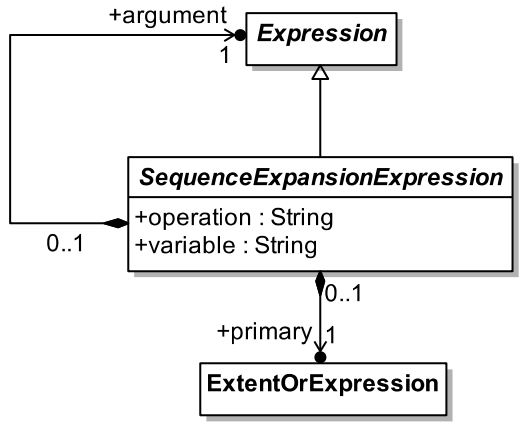

Figure 8.19 Abstract Syntax of Sequence Expansion Expressions

Cross References

1. Name see 7.6
2. Expression see 8.1
3. QualifiedName see 8.2
4. ExtentOrExpression see 8.3.17
5. SelectionOrRejectOperation see 8.3.20
6. CollectOrIterateOperation see 8.3.21
7. ForAllOrExistsOrOneOperation see 8.3.22
8. IsUniqueOperation see 8.3.23

Semantics

The operation name in a sequence expansion expression must be one of the reserved keywords listed in Table 8.3.

Table 8.3 Sequence Expansion Operations

Operation	Description
`select`	Select a sub-sequence of elements for which a condition is true.
`reject`	Select a sub-sequence of elements for which a condition is false.
`collect`	Concurrently apply a computation on each element of a sequence.
`iterate`	Sequentially apply a computation to each element of a sequence.
`forAll`	Test if a condition is true for all elements of a sequence.
`exists`	Test if a condition is true for at least one element of a sequence.
`one`	Test if a condition is true for exactly one element of a sequence
`isUnique`	Test if a computation has a different value for every element of a sequence.

Assignments made in the primary expression of a sequence expansion expression are available within its argument expression. In addition, the expansion variable is available as a local name within the argument expression, with the same type as the primary expression and multiplicity `[1..1]`. Its name must not conflict with any already assigned local name. Its assigned source is the sequence expansion expression itself.

For example, in the expression

```
c->select x (x>1)
```

"`select`" is the expansion operation name and "`x`" is the expansion variable name.

An expansion variable may not be reassigned within the argument expression of a sequence expansion expression, and it is considered unassigned after the sequence expansion expression. Further, while the argument expression may reference local names defined outside that expression, it may not reassign such local names. New local names may be defined and referenced within the argument expression, but all such names are considered unassigned after the sequence expansion expression.

NOTE. The above rule is necessary, since, if the sequence being expanded is empty, the argument expression will never be evaluated and names assigned within it would have no value on their assigned source. The rule also allows sequence expansion expressions to be mapped to expansion regions, which are not allowed to have outgoing flows or output pins in fUML (see fUML Specification, 7.4.5.2.2).

Even though a sequence expansion expression is not equivalent to a behavior invocation, a collection object may be directly acted on in such an expression analogously to how it may be acted on in a sequence operation expression (see 8.3.17). The operation `toSequence` is called on the collection object, and the expansion is applied to the result of that operation.

For example, if `customerList` has the type `List<Customer>`, then the sequence expansion expression

```
customerList->select c (c.name == customerName)
```

is equivalent to

```
Customer.toSequence()->select c (c.name == customerName)
```

Also as in a sequence operation expression (see 8.3.17), a sequence expansion expression allows the special notation in which the initial primary expression may be replaced with the name of a class to implicitly denote the the current extent of that class. Thus, the expression

```
Customer->select c (c.name == customerName)
```

is equivalent to

```
Customer.allInstances()->select c (c.name == customerName)
```

(where `Customer.allInstances()` is a class extent expression—see 8.3.14).

Specific semantics for each kind of sequence expansion operation are further discussed in subsequent subclauses.

8.3.20 `select` and `reject` Expressions

The `select` and `reject` operations are used in a sequence expansion expression (see 8.3.19) to specify a selection of elements from a sequence. The `select` operation is used to select elements that meet a given condition, while the `reject` operation is used to select those that do not meet a condition.

NOTE. The sequence expansion expression notation is not available at the minimum conformance level (see 2.2).

Examples

```
employees->select e (e.age>50)
```

```
employees->reject e (e.isMarried)
```

Syntax

```
SelectOrRejectOperation(e: SelectOrRejectExpression)
      = "select"(e.operation)
      | "reject"(e.operation)
```

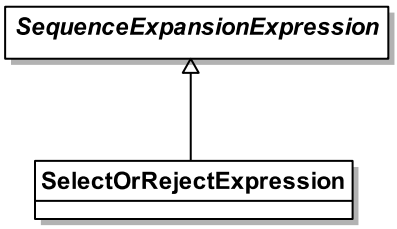

Figure 8.20 Abstract Syntax for `select` and `reject` Expressions

Cross References

1. SequenceExpansionExpression see 8.3.19

Semantics

A sequence expansion expression with a `select` or `reject` expression must have an argument expression with a type that conforms to type `Boolean` and a multiplicity upper bound of 1. The sequence expansion expression has the same type and multiplicity upper bound as its primary expression and a multiplicity lower bound of 0.

The `select` operation specifies a subset of a sequence. Each element for which the argument expression evaluates to true is included in the result sequence.

For example, the following expression selects all employees greater than 50 years old:

```
employees->select e (e.age>50)
```

The `reject` operation is similar to `select`, except that elements for which the argument expression evaluates to false are included in the result sequence. The `reject` operation is thus equivalent to a `select` operation with the argument expression negated.

For example, the following expression selects all employees who are *not* married:

```
employees->reject e (e.isMarried)
```

8.3.21 `collect` and `iterate` Expressions

The `collect` and `iterate` operations are used in a sequence expansion expression (see 8.3.19) to specify a sequence that is derived from some other sequence, but which may contain different elements than the original sequence. This computation is carried out concurrently in the case of the `collect` operation, but sequentially in the case of the `iterate` operation.

Examples

```
employees->collect e (e.birthDate)
```

```
processSteps->iterate step (step.execute())
```

Syntax

```
CollectOrIterateOperation(e: CollectOrIterateExpression)
      = "collect"(e.operation)
      | "iterate"(e.operation)
```

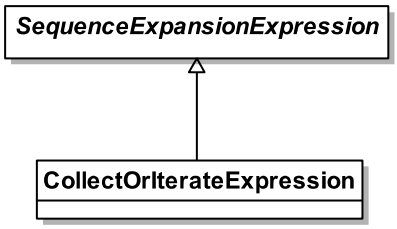

Figure 8.21 Abstract Syntax of `collect` and `iterate` expressions

Cross References

1. SequenceExpansionExpression see 8.3.19

Semantics

A sequence expansion expression with a `collect` or `iterate` operation may have an argument expression of any type. The sequence expansion expression has the same type as its argument expression and a multiplicity determined by multiplying the corresponding lower and upper bounds of the collection and argument expressions (where the product of the unbounded value `*` with anything is considered to be `*`).

The result of a `collect` or `iterate` operation is the sequence of the results of evaluating the argument expression for each element of the input sequence.

For example, the following expression results in the sequence of birth dates of all employees:

```
employees->collect e (e.birthDate)
```

Note that, when the argument expression is a property access expression, the collect operation is the same as a property access expression directly on the input sequence (see 8.3.6). Thus, the above example is equivalent to the simpler expression

```
employees.birthDate
```

An `iterate` operation has the same behavior as a `collect` operation, except that the argument expression is evaluated sequentially for all elements of the input sequence, in order, rather than concurrently, as is the case for `collect`. This can be useful when the argument expression potentially has side effects.

For example, in the evaluation of the expression

```
processSteps->iterate step (step.execute())
```

the execution of the process steps will occur sequentially, and the execution of each step will take place in the context resulting from the completion of the previous step. As for `collect`, any results returned from the `execute` operation invocations will be collected into a result sequence.

8.3.22 `forAll`, `exists` and `one` Expressions

The `forAll`, `exists` and `one` operations are used in a sequence expansion expression (see 8.3.19) to test a `Boolean` argument expression on the elements of a sequence. The `forAll` operation tests that the condition holds for all elements, the `exists` operation that it holds for at least one element and the `one` operation that it holds for exactly one element.

Examples

```
employees->forAll e (e.age<=65)
```

```
employees->exists e (e.firstName=="Jack")
```

```
employees->one e (e.title=="President")
```

Syntax

```
ForAllOrExistsOrOneOperation(e: ForAllOrExistsOrOneExpression)
      = "forAll"(e.operation)
      | "exists"(e.operation)
      | "one"(e.operation)
```

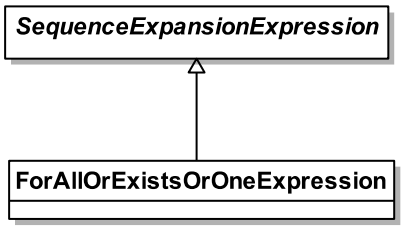

Figure 8.22 Abstract Syntax for `forAll`, `exists` and `one` Expressions

Cross References

1. SequenceExpansionExpression see 8.3.19

Semantics

A sequence expansion expression with a `forAll`, `exists` or `one` operation must have an argument expression with a type that conforms to type `Boolean` and a multiplicity upper bound of 1. The sequence expansion expression has type `Boolean` and multiplicity `[1..1]`.

The result of a `forAll` operation is true of the argument expression evaluates to true for all elements of the input sequence and false otherwise.

For example, the following expression evaluates to true if every employee is no older than 65:

```
employees->forAll e (e.age<=65)
```

The result of an `exists` operation is true if the argument expression evaluates to true for at least one element of the input sequence.

For example, the following expression evaluates to true if at least one employee has the first name `"Jack"`:

```
employees->exists e (e.firstName=="Jack")
```

The result of a `one` operation is true if the argument expression evaluates to true for exactly one element of the input sequence.

For example, the following expression evaluates to true if there is exactly one employee with the title `"President"`:

```
employees->one e (e.title=="President")
```

8.3.23 `isUnique` Expression

The `isUnique` operation is used in a sequence expansion expression (see 8.3.19) to test whether an expression evaluates to a different value for every element of a collection.

Examples

```
employees->isUnique e (e.employeeIdentificationNumber)
```

Syntax

```
IsUniqueOperation(e: IsUniqueExpression)
      = "isUnique"(e.operation)
```

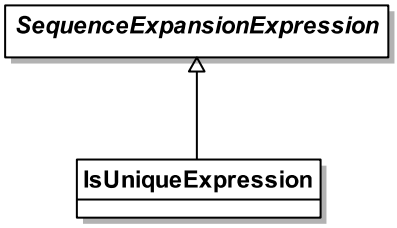

Figure 8.23 Abstract Syntax for `isUnique` Expressions

Cross References

1. SequenceExpansionExpression see 8.3.19

Semantics

A sequence expansion expression with an `isUnique` operation must have a single argument expression with a multiplicity upper bound of 1. The sequence expansion expression has type `Boolean` and multiplicity `[1..1]`.

The result of an `isUnique` operation is true if the argument expression evaluates to a different value for every element of the input collection.

For example, the following expression evaluates to true if every employee has a unique employee identification number:

```
employees->isUnique e (e.employeeIdentificationNumber)
```

8.4 Increment and Decrement Expressions

An *increment expression* is one that uses the increment operator `++`. A *decrement expression* is one that uses the decrement operator `--`. Either of these operators may be used in either a *prefix* form, in which the operator appears before the operand expression, or a *postfix* form, in which the operator appears after the operand expression.

Examples

Postfix Form

```
count++
size--
total[i]++
```

Prefix Form

```
++count
--numberWaiting[queueIndex]
```

Syntax

```
IncrementOrDecrementExpression(e: IncrementOrDecrementExpression)
      = PostfixExpression(e)
      | PrefixExpression(e) (e.isPrefix=true)
PostfixExpression(e: IncrementOrDecrementExpression)
      = LeftHandSide(e.operand) AffixOperator(e.operator)
PrefixExpression(e: IncrementOrDecrementExpression)
      = AffixOperator(e.operator) LeftHandSide(e.operand)
AffixOperator(op: String)
      = "++"(op) | "--"(op)
```

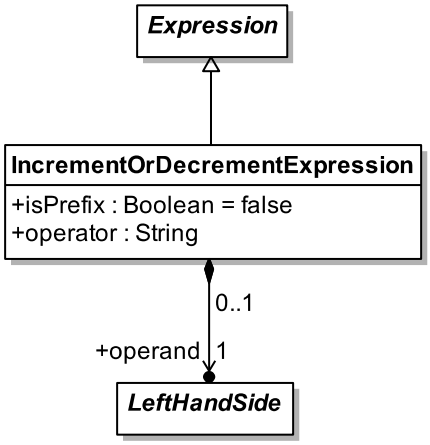

Figure 8.24 Increment and Decrement Expressions

Cross References

1. Expression see 8.1
2. LeftHandSide see 8.8

Semantics

The operand expression for an increment or decrement expression must conform to the syntax and static semantics for the left hand side of an assignment (see 8.8). It must have a type that conforms to type `Integer` and multiplicity upper bound of 1. The increment or decrement expression has type `Integer`, the same multiplicity lower bound as its operand expression and a multiplicity upper bound of 1.

The effect of an increment or decrement expression is to increment (`++`) or decrement (`--`) the value of its operand and then reassigns the result to the operand. If the operator is used as a postfix, then the value of the expression is the value of its operand *before* it is reassigned. If the operator is used as a prefix, the value of the expression is the values of its operand *after* it is reassigned.

For example, if the local name `a` has the value 5, then both `a++` and `++a` assign the value 6 to `a`. However the value of the expression `a++` itself is 5, while the value of `++a` is 6.

8.5 Unary Expressions

8.5.1 Overview

A *unary expression* is an expression with a single operand expression and an operator that performs some action on the values produced by the operand. Unary operators include numeric unary operators, Boolean negation and isolation. Cast expressions, which filter a sequence of values based on type, are

also considered unary expressions in terms of concrete syntax, even though their "operator" is given by a type name, not a fixed symbol.

The static and execution semantics of each kind of unary expression are discussed further in subsequent subclauses.

Syntax

```
UnaryExpression(e: Expression)
      = PrimaryExpression(e)
      | IncrementOrDecrementExpression(e)
      | BooleanUnaryExpression(e)
      | BitStringUnaryExpression(e)
      | NumericUnaryExpression(e)
      | CastExpression(e)
      | IsolationExpression(e)
```

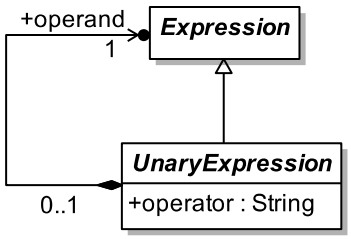

Figure 8.25 Base Abstract Syntax for Unary Expressions

Cross References

1. Expression see 8.1
2. PrimaryExpression see 8.3.1
3. IncrementOrDecrementExpression see 8.4
4. BooleanUnaryExpression see 8.5.2
5. NumericUnaryExpression see 8.5.4
6. CastExpression see 8.5.5
7. IsolationExpression see 8.5.6

8.5.2 `Boolean` Unary Expressions

A `Boolean` *unary expression* is a unary expression whose operator acts on and produces `Boolean` values. The only `Boolean` unary operator is the negation operator `!`.

Examples

```
!isActive
```

```
!this.running
```

Syntax

```
BooleanUnaryExpression(e: BooleanUnaryExpression)
      = "!"(e.operator) UnaryExpression(e.operand)
```

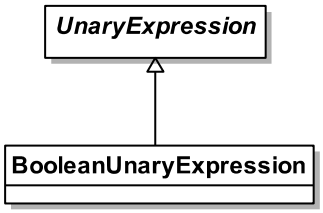

Figure 8.26 Abstract Syntax of `Boolean` Unary Expressions

Cross References

1. UnaryExpression see 8.5.1

Semantics

A `Boolean` unary expression must have an operand expression with a type that conforms to type `Boolean` and multiplicity `[1..1]`. The `Boolean` unary expression has type `Boolean` and multiplicity `[1..1]`.

The functionality of the `Boolean` negation operator is the same as an application of the Alf library `BooleanFunctions::'!'` function (see 11.4.1) with the operand expression as its argument. If the operand is true, the result is false. If the operand is false, the result is true.

8.5.3 `BitString` Unary Expressions

A *`BitString` unary expression* is a unary expression whose operator acts on a bit string (or an integer convertible to a bit string) and produces a bit string. The only `BitString` unary operator is the bit-wise complement operator `~`.

Examples

```
~registerContext
```

```
~memory.getByte(address)
```

Syntax

```
BitStringUnaryExpression(e: BitStringUnaryExpression)
      = "~"(e.operator) UnaryExpression(e.operand)
```

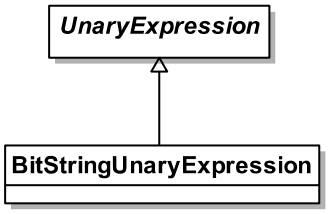

Figure 8.27 Abstract Syntax of `BitString` Unary Expressions

Semantics

A `BitString` unary expression must have an operand expression with a type that conforms to type `BitString` or `Integer` and multiplicity `[1..1]`. The `BitString` unary expression has type `BitString` and multiplicity `[1..1]`. If the operand is an integer, then it is first converted to a bit string by applying the library `BitStringFunctions::ToBitString` function (see 11.4.6).

The functionality of the `BitString` bit-wise complement operator is the same as an application of the Alf library `BitStringFunctions::'~'` function (see 11.4.6) with the operand expression as its argument. The result is a bit string that has a bit set (value 1) in each bit position in which the operand bit string had its bit unset (value 0) and a bit unset (value 0) in each bit position in which the operand bit string had its bit set (value 1).

8.5.4 Numeric Unary Expressions

A *numeric unary expression* is a unary expression that acts on and produces numeric values. The numeric unary operators are `+` and `-`.

Examples

```
+1234

-42

+(a*b)

-absoluteValue
```

Syntax

```
NumericUnaryExpression(e: NumericUnaryExpression)
      = NumericUnaryOperator(e.operator) UnaryExpression(e.operand)
NumericUnaryOperator(op: String)
      = "+"(op) | "-"(op)
```

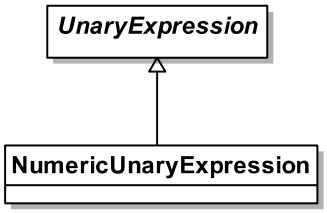

Figure 8.28 Abstract Syntax of Numeric Unary Expressions

Cross References

1. UnaryExpression see 8.5.1

Semantics

A numeric unary expression must have an operand expression with a type that conforms to type `Integer` or `Real` and a multiplicity upper bound of 1. A numeric unary expression has type `Integer` if its operand is of type `Integer` and type `Real` if its operation is of type `Real`. It has the same multiplicity lower bound as its operand expression and a multiplicity upper bound of 1.

The unary plus operator does not change its operand value, while the unary minus operator negates it. The unary minus operator has the same functionality as application of the Alf library `IntegerFunctions::Neg` function (see 11.4.2), if the operand is of type `Integer`, or `RealFunctions::Neg` function (see 11.4.3), if the operand is of type `Real`, with the operand expression as the argument.

NOTE. While the unary plus operator does not have any mathematical effect on its operand, it can be used as a way to effectively denote an `Integer` literal value. For example the literal "`1234`" has the type

`Natural` (see 7.8.3) and could be either an `Integer` or an `UnlimitedNatural` value. However the expression "`+1234`" is unambiguously an `Integer`.

8.5.5 Cast Expressions

A *cast expression* is used to filter the values of its operand expression to those of a given type. The type is named within parentheses and prefixes the operand expression as an effective unary operator.

Examples

```
(fUML::Syntax::Activity)this.getTypes()
```

```
(Person)invoice.payingParty
```

```
(any)this
```

Syntax

```
CastExpression(e: CastExpression)
      = "(" TypeName(e.typeName) ")" NonNumericUnaryExpression(e.operand)
NonNumericUnaryExpression(e: Expression)
      = PrimaryExpression(e)
      | PostfixExpression(e)
      | BooleanUnaryExpression(e)
      | BitStringUnaryExpression(e)
      | CastExpression(e)
      | IsolationExpression(e)
```

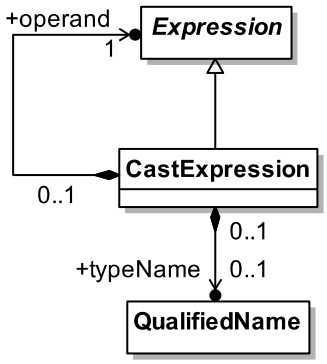

Figure 8.29 Abstract Syntax of Cast Expressions

Cross References

1. Expression — see 8.1
2. TypeName — see 8.2
3. QualifiedName — see 8.2
4. PrimaryExpression — see 8.3.1
5. TypeName — see 8.3.15
6. PostfixExpression — see 8.4
7. BooleanUnaryExpression — see 8.5.2
8. IsolationExpression — see 8.5.6

Semantics

Unless the type name in a cast expression is the keyword `any`, the cast expression has the given type (see also 8.2 on type names). If the type name is `any`, then the cast expression is untyped. If the type name is a qualified name, then it must resolve to a classifier, which must not be a template, though it may be a binding of a template classifier.

A cast expression is evaluated by first evaluating its operand expression, producing a sequence of values. Any values of the operand expression whose dynamic type does not conform to the type of the cast expression are filtered out, so that all result values of the cast expression are of the given type. If the cast expression is untyped, then no values are filtered out.

For example, the cast expression

```
(Integer)any[]{1,"banana",2}
```

evaluates to

```
Integer[]{1,2}
```

The library type `Natural` is a subtype of `Integer` and `UnlimitedNatural`. This means that natural literals of type `Natural` (see 7.8.3) can be cast to `Integer` or `UnlimitedNatural`. Thus, `(Integer)2` is the `Integer` value 2, while `(UnlimitedNatural)2` is the `UnlimitedNatural` value 2.

In addition, for the purpose of a cast expression, any non-negative `Integer` value is considered to conform to the type `UnlimitedNatural`, and vice versa. `Integer` values are converted to `UnlimitedNatural` values using the Alf library `IntegerFunctions::ToUnlimitedNatural` function (see 11.4.2). `UnlimitedNatural` values are converted to `Integer` values using the Alf library `UnlimitedNaturalFunctions::ToInteger` function (see 11.4.5).

For example, the cast expression

```
(Integer)UnlimitedNatural[]{1,2,*}
```

evaluates to

```
Integer[]{1,2}
```

and the cast expression

```
(UnlimitedNatural)Integer[]{-1,0,1}
```

evaluates to

```
UnlimitedNatural[]{0,1}
```

Further, any `Integer` value is considered to conform to the type `BitString` and vice versa. `Integer` values are converted to `BitString` values using the Alf library `BitStringFunctions::ToBitString` function and `BitString` values are converted to `Integer` values using the `BitStringFunctions::ToInteger` function (see 11.4.6).

Finally, any `Integer` value is also considered to conform to the type `Real` and vice versa. `Integer` values are converted to `Real` values using the Alf library `IntegerFunctions::ToReal` function (see 11.4.2) and `Real` values are converted to `Integer` values using the `RealFunctions::ToInteger` function (see 11.4.3).

The multiplicity lower bound of a cast expression is 0, unless one of the following conditions hold, in which case the multiplicity lower bound is the same as that of its operand expression.

- The cast expression is untyped.
- The type of the cast expression is `Integer` and the type of its operand expression is `BitString` or `Real`.
- The type of the cast expression is `BitString` or `Real` and the type of its operand expression is `Integer`.
- The type of the operand expression of the case expression conforms to the type of the cast expression.

The multiplicity upper bound of a cast expression is the same as that of its operand expression.

8.5.6 Isolation Expressions

An *isolation expression* is a unary expression with the isolation operator `$`.

NOTE. The isolation expression notation is not available at the minimum conformance level (see 2.2).

Examples

```
$this.monitor.getActiveSensor().getReading()
```

Syntax

```
IsolationExpression(e: IsolationExpression)
      = "$"(e.operator) UnaryExpression(e.operand)
```

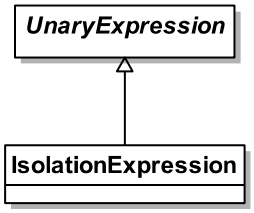

Figure 8.30 Abstract Syntax of Isolation Expressions

Cross References

1. UnaryExpression see 8.5.1

Semantics

The isolation operator indicates that its operand expression should be evaluated in isolation, similarly to the use of the `@isolated` annotation for a statement (see 9.2). That is, during the evaluation of the operand expression, no object accessed as part of the evaluation of the expression or as the result of a synchronous invocation from the expression may be modified by any action that is not executed as part of the operand expression or as the result of a synchronous invocation from that expression.

NOTE. See 8.5.4.1 of the fUML Specification for a complete discussion of the semantics of isolation.

An isolation expression has the type and multiplicity of its operand expression.

8.6 Binary Expressions

8.6.1 Overview

A *binary expression* is an expression with two operand expressions and an operator that performs some action on the values produced by the operands. Binary operators include arithmetic, relational, equality, logical and conditional logical operators. Classification expressions, which test if a value has a certain type, are also syntactically similar to binary expressions, except that one of the "operands" of a classification expression is actually a type name.

Syntax

See the concrete syntax for each kind of binary expression in subsequent subclauses.

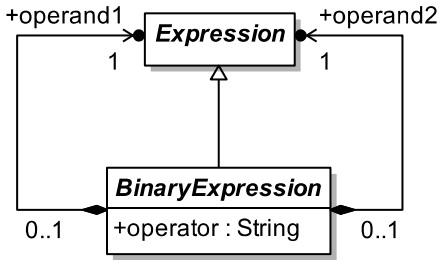

Figure 8.31 Base Abstract Syntax for Binary Expressions

Semantics

Except in the case of conditional logical expressions (see 8.6.8), the operand expressions of a binary expression are evaluated *concurrently* and then the operator is applied to their results. Because of the concurrent evaluation of the operands, it is not legal to assign the same local name in both operand expressions.

The semantics of each kind of binary expression are discussed further in subsequent subclauses.

8.6.2 Arithmetic Expressions

An *arithmetic expression* is a binary expression with an arithmetic operator. Arithmetic operators include the *multiplicative operators* are `*`, `/` and `%` and the *additive operators* `+` and `-`. The multiplicative operators all have a higher precedence than the additive operators. All arithmetic operators are syntactically left-associative (they group from left to right).

Examples

```
amount * interestRate

duration / timeStep

length % unit

initialPosition + positionChange

basePrice - discount
```

Syntax

```
UnaryOrMultiplicativeExpression(e: Expression)
      = UnaryExpression(e)
      | MultiplicativeExpression(e)
```

```
MultiplicativeExpression(e: ArithmeticExpression)
      = UnaryOrMultiplicativeExpression(e.operand1)
        MultiplicativeOperator(e.operator) UnaryExpression(e.operand2)
MultiplicativeOperator(op: String)
      = "*"(op) | "/"(op) | "%"(op)
UnaryOrArithmeticExpression(e: Expression)
      = UnaryOrMultiplicativeExpression(e)
      | AdditiveExpression(e)
AdditiveExpression(e: ArithmeticExpression)
      = UnaryOrArithmeticExpression(e.operand1)
        AdditiveOperator(e.operator)
        UnaryOrMultiplicativeExpression(e.operand2)
AdditiveOperator(op: String)
      = "+"(op) | "-"(op)
```

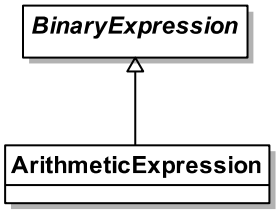

Figure 8.32 Abstract Syntax of Arithmetic Expressions

Cross References

1. UnaryExpression see 8.5.1
2. BinaryExpression see 8.6.1

Semantics

The operand expressions for an arithmetic operator other than `+` or `%` must be of a type that conforms to type `Integer` or `Real`. The operand expressions for the `+` operator must either both have types that conform to type `Integer` or `Real` or both have types that conform to type `String`. The operand expressions for the `%` operator must both be of type `Integer`. In all cases, the operand expressions must have a multiplicity upper bound of 1.

The functionality of the arithmetic operators is equivalent to the application of the similarly named primitive functions from the library `IntegerFunctions` package (see 11.4.2), `RealFunctions` package (see 11.4.3) or `StringFunctions` package (see 11.4.4), depending on the type of the operand expressions, with the two operand expressions as arguments. If one of the operand expressions is of type `Real` and the other is of type `Integer`, then the primitive function from the `RealFunctions` package is used, with real conversion performed on the `Integer` operand (see 8.8).

The `*` operator denotes multiplication, the `/` operator denotes division and the `%` operator denotes remainder. The `+` operator denotes either addition or string concatenation. The `-` operator denotes subtraction.

An arithmetic expression has the same type as its operands and a multiplicity upper bound of 1. Its multiplicity lower bound is 0 if its operator is `/` or if the lower bound if either operand expression is 0 and 1 otherwise.

NOTE. A division by zero results in no value, so a division operation has a multiplicity lower bound of 0.

8.6.3 Shift Expressions

A *shift expression* is a binary expression with a shift operator. The *shift operators* are left shift <<, signed right shift >> and unsigned right shift >>>. They have a lower precedence than any of the arithmetic operators. They are syntactically left-associative (they group left to right). The first operand of a shift operator must be a bit string (or an integer convertible to a bit string) and the second operand then specifies the number of bit positions that bit string is to be shifted.

Examples

```
bitmask << wordLength
```

```
wordContent >> offset
```

```
(value&0xF0) >>> 8
```

Syntax

```
ArithmeticOrShiftExpression(e: Expression)
      = UnaryOrArithmeticExpression(e)
      | ShiftExpression(e)
ShiftExpression(e: ShiftExpression)
      = ArithmeticOrShiftExpression(e.operand1) ShiftOperator(e.operator)
        UnaryOrArithmeticExpression(e.operand2)
ShiftOperator(op: String)
      = "<<"(op) | ">>"(op) | ">>>"(op)
```

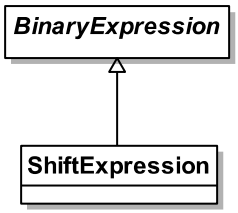

Figure 8.33 Abstract Syntax of Shift Expressions

Cross References

1. BinaryExpression see 8.6.1
2. UnaryOrArithmeticExpression see 8.6.2

Semantics

The first operand expression of a shift expression must have a type that conforms to the type `BitString` or `Integer`. The second operand expression must have a type that conforms to the type `Integer`. The operand expressions must each have multiplicity `[1..1]`.

The shift operators are used to perform bit shifts on bit strings. If the first operand is an integer, then it is first converted to a bit string by applying the library `BitStringFunctions:: ToBitString` function (see 11.4.6). High-order bits in a bit string are considered to be on the left, while low-order bits are on the right.

The functionality of the shift operators is equivalent to the application of the similarly named primitive functions from the library `BitStringFunctions` package (see 11.4.6) with the two operand expressions as arguments.

The value of `b<<n` is `b` left-shifted n positions. Since the bit string length is fixed, the left `n` bits of `b` are lost. The right `n` bits of the resulting value are zero.

The value of `b>>n` is `b` right-shifted n positions. Since the bit string length is fixed, the right `n` bits of `b` are lost. The left `n` bits of the resulting value are set the same as the rightmost (highest bit position) bit of `b`. If `b` is the twos-complement representation of an integer (e.g., if it was converted from an integer value), then this corresponds to sign-extension of the original value.

The value of `b>>>n` is the same as for `b>>n`, except that the left n bits of the resulting value are zero (zero-extension instead of sign-extension).

A shift expression has type `BitString` and multiplicity `[1..1]`.

8.6.4 Relational Expressions

A *relational expression* is a binary expression with a relational operator. The *relational operators* are `<`, `>`, `<=` and `>=`. They have a lower precedence than any of the arithmetic or shift operators. The relational operators are *not* associative, and it is not legal to use more than one in an expression without parentheses. For example, `a<b<c` is not syntactically legal, though `(a<b)<c` and `a<(b<c)` are. (But, even with parentheses, these expressions are not actually useful, since the parenthesized expression has type `Boolean`, which is not legal as an argument to the outer `<` operator.)

NOTE. The restriction on associative relational expressions avoids a syntactic ambiguity with the syntax for the invocation of a template behavior with an explicit binding. For example, the expression `A<B>(C)` can be unambiguously parsed as an invocation of the behavior `A<B>`, since the "`>`" cannot legally be parsed as a relational greater than operator.

Examples

```
sensorReading > threshold
```

```
size < maxSize
```

```
size >= minSize
```

```
count <= limit
```

Syntax

```
ShiftOrRelationalExpression(e: Expression)
      = ArithmeticOrShiftExpression(e)
      | RelationalExpression(e)
RelationalExpression(e: RelationalExpression)
      = ArithmeticOrShiftExpression(e.operand1)
        RelationalOperator(e.operator)
        ArithmeticOrShiftExpression(e.operand2)
RelationalOperator(op: String)
      = "<"(op) | ">"(op) | "<="(op) | ">="(op)
```

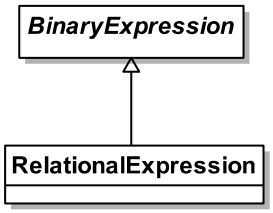

Figure 8.34 Abstract Syntax of Relational Expressions

Cross References

1. BinaryExpression see 8.6.1
2. ArithmeticOrShiftExpression see 8.6.3

Semantics

The operand expressions for a comparison operator must `both have a type that conforms to type Natural, Integer or Real, or both have a type that conforms to type Natural or UnlimitedNatural`. That is, it is not allowed to have one operand expression be `Integer` or `Real` and the other be `UnlimitedNatural`. The operand expressions must each have a multiplicity upper bound of 1.

The relational operators are used to compare the values of two numbers. The number being compared may be either integers, reals or unlimited naturals, but it is not legal to directly compare an integer or real to an unlimited natural number.

Thus, the expression

```
3 < *
```

is legal (and evaluates to true), since the natural literal 3 is automatically cast to `UnlimitedNatural` in this context. However, the expression

```
+3 < *
```

is not legal, because the literal `+3` has type `Integer`. A cast expression (see 8.5.5) must be used in order to directly compare an integer value to an unlimited natural value; for example,

```
(UnlimitedNatural)(+3) < *
```

evaluates to true.

The functionality of the relational operators is equivalent to the application of the similarly named primitive functions from the library `IntegerFunctions` package (see 11.4.2), RealFunctions package (see 11.4.3) or `UnlimitedFunctions` package (see 11.4.5), depending on the type of the operand expressions, with the two operand expressions as arguments. If one of the operand expressions is of type `Real` and the other is of type `Integer`, then the primitive function from the `RealFunctions` package is used, with real conversion (see 8.8) performed on the `Integer` operand.

NOTE. The Alf Standard Model Library comparison functions are based on the comparison functions available in the fUML Foundation Model Library. The Foundation Model Library does not provide comparison operators for the primitive type `String`.

A comparison expression has type `Boolean` and a multiplicity upper bound of 1. Its multiplicity lower bound is 0 if the lower bound if either operand expression is 0 and 1 otherwise.

8.6.5 Classification Expressions

A *classification expression* is an expression with a single operand expression followed by one of the *classification operators* `instanceof` or `hastype`. A classification expression is used to determine whether the result of its operand expression has a certain type, which is given as a qualified name after the classification operator.

Examples

```
action instanceof ActionActivation
```

```
'signal' hastype SignalArrival
```

Syntax

```
RelationalOrClassificationExpression(e: Expression)
      = ArithmeticOrRelationalExpression (e)
      | ClassificationExpression(e)
ClassificationExpression(e: ClassificationExpression)
      = ArithmeticOrRelationalExpression(e.operand)
        ClassificationOperator(e.operator)
        QualifiedName(e.typeName)
ClassificationOperator(op: String)
      = "instanceof"(op) | "hastype"(op)
```

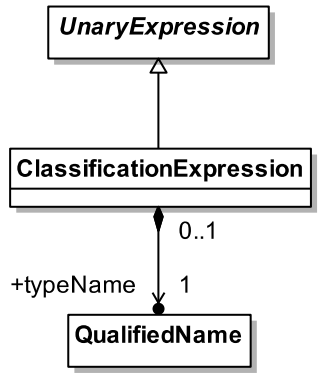

Figure 8.35 Abstract Syntax of Classification Expressions

NOTE. The concrete syntax for classification expressions, in terms of form and precedence, is similar to that of a normal binary expression, except that one of the "operands" is a type rather than a second expression. However, since only one operand is actually an expression, in the abstract syntax a classification expression is treated as a kind of unary expression.

Cross References

1. QualifiedName see 8.2
2. UnaryExpression see 8.5.1
3. ArithmeticOrRelationalExpression see 8.6.4

Semantics

The qualified name in a classification expression must resolve to a classifier. The classifier must not be a template, though it may be the binding of a template classifier (see 8.2). However, if one operand expression is of type `Integer` and the other is of type `Real`, then real conversion (see 8.8) is performed

on the `Integer` operand and the resulting real value is used for the comparison. Thus, `1 == 1.0`, for example, evaluates to true.

The operand expression must have multiplicity `[1..1]`. A classification expression has type `Boolean` and multiplicity `[1..1]`.

The `instanceof` operator checks if the dynamic type of its operand conforms to the given type—that is, whether the argument type is the same as or a direct or indirect subclass of the given type. The `hastype` operator, on the other hand, checks if the dynamic type of its argument is the same as the given type.

8.6.6 Equality Expressions

An *equality expression* is a binary expression with one of the *equality operators* `==` (equal to) or `!=` (not equal to). The equality operators are analogous to the relational operators, except for their lower precedence. They are syntactically left-associative (they group left to right), so `a==b==c` parses as `(a==b)==c`. However, the result type of `a==b` is `Boolean`, so if `c` is not `Boolean`, the expression will always be `false`. Thus, `a==b==c` does *not* test whether `a`, `b` and `c` are all equal.

Examples

```
errorCount==0
```

```
nextRecord!=endRecord
```

```
list.next==null
```

Syntax

```
ClassificationOrEqualityExpression(e: Expression)
      = RelationalOrClassificationExpression(e)
      | EqualityExpression(e)
EqualityExpression(e: BinaryExpression)
      = ClassificationOrEqualityExpression(e.operand1)
        EqualityOperator(e.operator)
        RelationalOrClassificationExpression(e.operand2)
EqualityOperator(op: String)
      = "=="(op) | "!="(op)
EqualityOrAndExpression(e: Expression)
      = ClassificationOrEqualityExpression(e)
      | AndExpression(e)
```

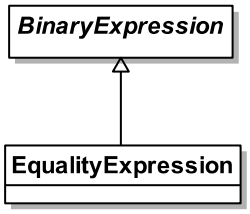

Figure 8.36 Abstract Syntax of Equality Expressions

Cross References

1. BinaryExpression see 8.6.1
2. RelationalOrClassificationExpression see 8.6.5

Semantics

For primitive types, the equality operators test whether two values of the same type are the same value. A value of a primitive type is never equal to a value of any other type.

NOTE. Since `String` is a primitive type in UML, *not* a class, strings may be tested for equality of *value* using the regular equality operators.

For structured data types, the equality operators test whether two values of the same type have equal values for each corresponding attribute. A value of one structured data type is never equal to a value of any other type. Note also that the participation of a value in association links does *not* affect its equality to other values.

For classes, the equality operators test whether the argument values are references to the *same* object, that is, they test object identity.

Further, if one of the arguments to an equality operator is empty (that is, it is the empty sequence), then it is considered to be equal to another empty value, but unequal to any non-empty value. Thus, `a==null` is true if and only if `isEmpty(a)`.

An operand expression for an equality operator must have a multiplicity upper bound of 1. An equality expression has type `Boolean` and multiplicity `[1..1]`.

8.6.7 Logical Expressions

A *logical expression* is a binary expression with one of the *logical operators,* including the *and* operator `&`, the *exclusive or* operator `^` and the *inclusive or* operator `|`. These operators may also be used to perform *bit-wise* logical operations on bit strings (or integers convertible to bit strings). The logical operators have different precedence, with `&` having the highest precedence and `|` having the lowest precedence. They all have lower precedence than the equality operators.They are syntactically left-associative (group left-to-right) and commutative (if their argument expressions have no side effects).

Examples

```
sensorOff | sensorError
```

```
i > min & i < max | unlimited
```

```
bitString ^ mask
```

```
registerContent & 0x00FF
```

Syntax

```
EqualityOrAndExpression(e: Expression)
      = ClassificationOrEqualityExpression(e)
      | AndExpression(e)
AndExpression(e: LogicalExpression)
      = EqualityOrAndExpression(e.operand1) "&"(e.operator)
        ClassificationOrEqualityExpression(e.operand2)
AndOrExclusiveOrExpression(e: Expression)
      = EqualityOrAndExpression(e)
      | ExclusiveOrExpression(e)
ExclusiveOrExpression(e: LogicalExpression)
      = AndOrExclusiveOrExpression(e.operand1) "^"(e.operator)
        EqualityOrAndExpression(e.operand2)
ExclusiveOrOrInclusiveOrExpression(e: Expression)
      = AndOrExclusiveOrExpression(e)
      | InclusiveOrExpression(e)
```

```
InclusiveOrExpression(e: LogicalExpression)
      = ExclusiveOrOrInclusiveOrExpression(e.operand1) "|"(e.operator)
        AndOrExclusiveOrExpression(e.operand2)
```

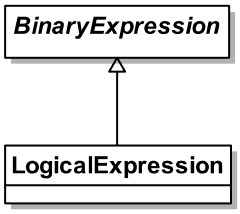

Figure 8.37 Abstract Syntax of Logical Expressions

Cross References

1. BinaryExpression see 8.6.1
2. ClassificationOrEqualityExpression see 8.6.6

Semantics

The operand expressions for a logical operator must be of a type that conforms to type `Boolean`, `BitString` or `Integer`. However, if one of the operands is `Boolean`, then the other must also be `Boolean`. The operand expressions must each have multiplicity `[1..1]`. Any operand that is an integer is converted to a bit string by applying the library `BitStringFunctions::ToBitString` function (see 11.4.6).

The functionality of the logical operators is equivalent to the application of the similarly named primitive functions from the library `BooleanFunctions` package (see 11.4.1), if the operands have type `Boolean`, or `BitStringFunctions` package (see 11.4.6), with the two operand expressions as arguments.

For `Boolean` operands, the logical operators perform the indicated logical operation and produce a `Boolean` result:

- For `&`, the result value is true if both argument values are true; otherwise, the result is false.
- For `^`, the result value is true if the argument values are different; otherwise, the result is false.
- For `|`, the result value is false if both argument values are false; otherwise, the result is true.

For `BitString` or `Integer` operands, the logical operators perform the indicated logical operation, as above, bit-wise on corresponding bits of the operands and produce a `BitString` result. For the purpose of carrying out bit-wise logical operations, a bit that is set (value of 1) is considered to be "true" while a bit that is unset (value of 0) is considered to be "false".

A logical expression has the same type as its operand expressions and multiplicity `[1..1]`.

8.6.8 Conditional Logical Expressions

A *conditional logical expression* is a binary expression using one of the *conditional logical operators* `&&` (conditional-and) and `||` (conditional-or). The operators are similar to the logical operators `&` and `|`, except that the evaluation of their second operand expression is conditioned on the result of evaluating the first expression. In the case of the `&&` operator, the second operand is evaluated only if the value of

the first operand is true. In the case of the `||` operator, the second operand is evaluated only if the value of the first operand is false.

Examples

```
index > 0 && value[index] < limit
```

```
index == 0 || value[index] >= limit
```

Syntax

```
InclusiveOrOrConditionalAndExpression(e: Expression)
      = ExclusiveOrOrInclusiveOrExpression(e)
      | ConditionalAndExpression(e)
ConditionalAndExpression(e: ConditionalLogicalExpression)
      = InclusiveOrOrConditionalAndExpression(e.operand1) "&&"(e.operator)
        ExclusiveOrOrInclusiveOrExpression(e.operand2)
ConditionalAndOrConditionalOrExpression(e: Expression)
      = InclusiveOrOrConditionalAndExpression(e)
      | ConditionalOrExpression(e)
ConditionalOrExpression(e: ConditionalLogicalExpression)
      = ConditionalAndOrConditionalOrExpression(e.operand1) "||"(e.operator)
        InclusiveOrOrConditionalAndExpression(e.operand2)
```

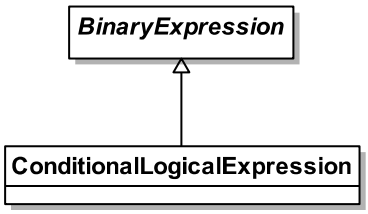

Figure 8.38 Abstract Syntax of Conditional Logical Expressions

Cross References

1. BinaryExpression see 8.6.1
2. ExclusiveOrOrInclusiveOrExpression see 8.6.7

Semantics

An expression with a conditional-and operator is evaluated by first evaluating its first operand expression. If the first operand expression evaluates to true, then the second operand expression is evaluated, and its value is the value for the conditional-and expression. If the first operand expression evaluates to false, however, the conditional-and expression evaluates to false without ever evaluating its second operand expression.

An expression with a conditional-or operator is evaluated by first evaluating its first operand expression. If the first operand expression evaluates to false, then the second operand expression is evaluated, and its value is the value for the conditional-or expression. If the first operand expression evaluates to true, however, the conditional-or expression evaluates to true without ever evaluating its second operand expression.

Since, if the second operand expression is evaluated at all, it is always evaluated after the first, a conditional logical operator expression is fully associative, even in the presence of side effects. That is, not only does an expression of the form `(expr1 && expr2) && expr3` produce the same result as `expr1 && (expr2 && expr3)`, the subexpression `expr1`, including any side effects, will always be

evaluated before *expr2*, and *expr2* before *expr3*. Similarly, an expression of the form `(expr1 || expr2) || expr3` produces the same result as `expr1 || (expr2 || expr3)`, with the subexpression *expr1* always evaluated before *expr2*, and *expr2* before *expr3*. Note that this is *not* guaranteed for the logical operators `&` and `|`, for which the operand expressions are evaluated concurrently.

The operand expressions to a conditional logical expression must be of a type that conforms to type `Boolean` and have multiplicity `[1..1]`. A conditional logical expression has type `Boolean` and multiplicity `[1..1]`.

Assignments made in the first operand expression of a conditional logical expression are available in the second operand expression, if it is evaluated. The multiplicity and type of local and parameter names reference in the second operand expression are adjusted based on the first operand expression evaluating to true, for a conditional-and expression, or false, for a conditional-or expression (see 8.7 on local name multiplicity and type adjustment).

Local names that are defined in either the first or the second operand expression are also available after the conditional logical expression. If a name is defined in the second operand expression, and that expression is not evaluated, then the name is defined but empty after the evaluation of the conditional logical expression.

8.6.9 Null-Coalescing Expressions

A *null-coalescing expression* is a binary expression using the *null-coalescing operator* `??`. The second operand expression of a null-coalescing expression is evaluated if the first operand expression does not produce a value (that is, it evaluates to `null`). The null-coalescing operator has a higher precedence than any other binary operator and it is syntactically right-associative (it groups right-to-left), so an expression of the form `a??b??c` is equivalent to `a??(b??c)` and will evaluate to the first of the values of `a`, `b` or `c` that is not `null`, or to `null` if all of `a`, `b` and `c` are `null`.

Examples

```
this.getSelectedDirectory() ?? this.defaultDirectory
```

```
Customer->select c (c.email == email) ?? new Customer(email)
```

```
list[2] ?? list[1]
```

```
WriteLine(content ?? "null");
```

Syntax

```
ConditionalLogicalOrNullCoalescingExpression(e: Expression)
      = ConditionalAndOrConditionalOrExpression(e)
      | NullCoalescingExpression(e)
NullCoalescingExpression(e: NullCoalescingExpression)
      = ConditionalAndOrConditionalOrExpression(e.operand1) "??"(e.operator)
        ConditionalLogicalOrNullCoalescingExpression(e.operand2)
```

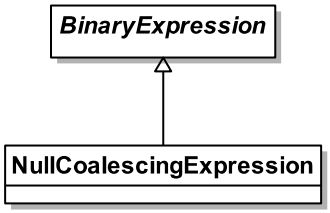

Figure 8.39 - Abstract Syntax of Null-Coalescing Expressions

Cross References

1. BinaryExpression see 8.6.1
2. ConditionalAndOrConditionalOrExpression see 8.6.8

Semantics

A null-coalescing expression is evaluated by first evaluating its first operand expression. If this expression produces at least one value, then these values are the result of the null-coalescing expression. If the evaluation of the first operand expression produces no values (i.e, it is equal to `null`), then the second operand expression is evaluated and its result is the result of the null-coalescing expression.

If either of the operand expressions of a null-coalescing expression is identically null (that is, it is untyped with multiplicity 0..0), then the type of the null-coalescing expression is the same as the type of the other operand expression. Otherwise, the type of a null-coalescing is the effective common ancestor (see 8.8) of the types of its operand expressions, if that exists, and it is untyped if no effective common ancestor exists.

The multiplicity lower bound of a null-coalescing expression is the same as that of its first operand expression, if that is not 0; otherwise it is 1, if the multiplicity lower bound of the second operand expression is not 0; otherwise it is 0. The multiplicity upper bound of the null-coalescing expression is the maximum of the multiplicity upper bounds of its two operand expressions.

Assignments made in the first operand expression of a null-coalescing expression are available in the second operand expression, if it is evaluated. Local names that are defined in the either the first or second operand expression are also available after the null-coalescing expression. If a name is defined in the second operand expression, and that expression is not evaluated, then the name is defined but empty after the evaluation of the null-coalescing expression.

NOTE. A null-coalescing expression of the form *`operand1 ?? operand2`* is equivalent to a conditional-test expression (see 8.7) of the form

```
(temp = operand1) != null? temp: operand2
```

where `temp` is some local name not otherwise used. Notice, in particular, that the first operand expression is only evaluated *once*, with the result of that evaluation being used both for the test against null and for the result of the overall expression, if it is not null.

8.7 Conditional-Test Expressions

A *conditional-test expression* uses the `Boolean` value of one expression to determine which of two other expressions should be evaluated. The conditional-test operator thus has *three* operands. The `?` symbol appears between the first and second operand expressions and `:` appears between the second and third expressions. The conditional-test operator is syntactically right-associative (it groups right-to-left), so that an expression of the form `a?b:c?d:e` is equivalent to `a?b:(c?d:e)`.

Examples

```
isNormalOps? readPrimarySensor(): readBackupSensor()
```

Syntax

```
ConditionalExpression(e: Expression)
      = ConditionalLogicalOrNullCoalescingExpression(e)
      | ConditionalTestExpression(e)
```

```
ConditionalTestExpression(e: ConditionalTestExpression)
      = ConditionalLogicalOrNullCoalescingExpression(e.operand1) "?"
        Expression(e.operand2) ":" ConditionalExpression(e.operand3)
```

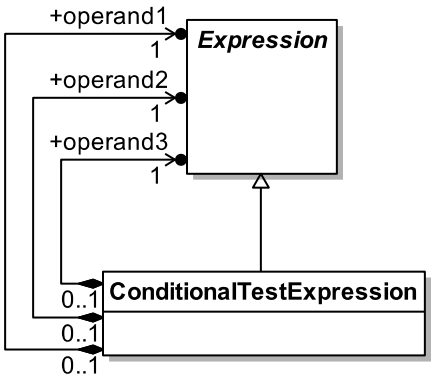

Figure 8.40 Abstract Syntax of Conditional-Test Expressions

Cross References

1. Expression see 8.1
2. ConditionalLogicalOrNullCoalescingExpression see 8.6.9

Semantics

A conditional test is evaluated by first evaluating the first operand expression. If this expression evaluates to true, then the second operand expression is evaluated, providing the values for the conditional-test expression. If the first operand expression evaluates to false, then the third operand expression is evaluated, providing the values for the conditional-test expression.

The first operand expression for a conditional-test operator must be of a type that conforms to type `Boolean` and have multiplicity `[1..1]`.

The type of a conditional-test operator expression is the effective common ancestor (see below) of the types of its second and third operand expressions, if one exists. If none exists, then the conditional-test operator expression is untyped.

The multiplicity lower bound of a conditional-test operator expression is zero if the multiplicity lower bound of its first operand expression is zero and the minimum of the multiplicity lower bounds of its second and third operand expressions otherwise. Its multiplicity upper bound is the maximum of the multiplicity upper bounds of its second and third operand expressions.

A conditional-test operator has special rules for assignment, since it evaluates its arguments sequentially. Local names assigned in the first operand expression may be used in the second and third operand expressions. The multiplicity and type of local and parameter names in the second operand expression are adjusted based on the first operand expression evaluating to true (see below on local name multiplicity and type adjustment). Similarly, the multiplicity and type of local and parameter names in the second operand expression are adjusted based on the first operand expression.

Further, any newly defined local names in the second and third argument expressions must be defined in both those expressions.

Effective common ancestor

A *common ancestor* for a set of classifiers is a classifier that is either equal to or a generalization (directly or indirectly) of all the classifiers in the given set.

A *most specialized common ancestor* is a common ancestor for which there is no other common ancestor, for the same given set of classifiers, that is a specialization (directly or indirectly) of the most specialized common ancestor. Due to multiple generalization, it is possible for a set of classifiers to have more than one most specialized common ancestor.

If a set of classifiers has a single most specialized common ancestor, then this is the *effective common ancestor* for the set. Otherwise, the effective common ancestor of the set of most specialized common ancestors (if any) is also the effective common ancestor of the original set of classifiers. Note that some sets of classifiers have no effective common ancestor.

Local name and multiplicity adjustment

The assigned source information for a local or parameter name may be *adjusted* if it is known to be null, non-null or of a certain type if a given Boolean expression is considered to be true or false (depending on the context), as follows:

- If the name is known to be null, its multiplicity lower bound is adjusted to 0.
- If the name is known to be non-null, its multiplicity lower bound is adjusted to 1.
- If the name is known to be of a certain type, its *known subtype* is set to that type (this will always be a subtype of the original implicitly or explicitly *declared type* of the name).

Whether a local or parameter name is known null, non-null or of a certain type is determined as given below.

The *relevant names* of an expression `E` are defined as follows:

- If `E` is a name expression (see 8.3.3) consisting of a single local or parameter name, then that is the single relevant name for `E`.
- If `E` is an assignment expression (see 8.8), then its relevant names include all the relevant names of its right-hand side, and, if its left-hand side is a local or parameter name, then that name is included, too.
- If `E` is a parenthesized expression (see 8.3.5), then its relevant names are all those of its contained expression.
- Otherwise, `E` has no relevant names.

For a Boolean expression of one of the following forms, all the relevant names of the contained expression `E` are *known to be null* if the containing expression evaluates to *true* or *known to be non-null* if the containing expression evaluates to *false*.

- A behavior invocation expression (see 8.3.9) of the form `IsEmpty(E)` or `isEmpty(E)`.
- A sequence operation expression (see 8.3.17) of the form `E->IsEmpty()` or `E->isEmpty()`.
- An equality expression (see 8.6.6) of the form `E == null`.

For a Boolean expression of one of the following forms, all the relevant names of the contained expression *E* are *known to be null* if the containing expression evaluates to *false* or *known to be non-null* if the containing expression evaluates to *true*.

- A behavior invocation expression (see 8.3.9) of the form `NotEmpty(E)` or `notEmpty(E)`.
- A sequence operation expression (see 8.3.17) of the form `E->NotEmpty()` or `E->notEmpty()`.
- An equality expression (see 8.6.6) of the form `E != null`.

In the above, `IsEmpty` and `NotEmpty` are primitive behaviors from the library package `SequenceFunctions` (see 11.4.7), and `isEmpty` and `notEmpty` are primitive behaviors from the library package `CollectionFunctions` (see 11.6). These primitive behaviors may also be referenced via qualified names or import aliases, as appropriate in the context of the containing expression.

For a classification expression (see 8.6.5) of the `E instanceof T` or `E hastype T`, all the relevant names of the contained expression `E` are *known to have type* `T` if the containing expression evaluates to *true*.

If a conditional-and expression (see 8.6.8 of the form `E1 && E2` evaluates to *true*, then any relevant name of `E1` is known to be null, non-null or of a certain type, if this can determined as given above assuming `E1` evaluates to true, and similarly for `E2`. If a conditional-or expression (see 8.6.8) of the form `E1 || E2` evaluates to *false*, then any relevant name of `E1` is known to be null, non-null or of a certain type, if this can determined as given above assuming `E1` evaluates to false, and similarly for `E2`.

For expressions other than one of the forms given above, no determination is made on whether any names referenced in the expression are null, non-null or of a certain type.

8.8 Assignment Expressions

An *assignment expression* is used to assign a value to a local name, output parameter or attribute. There are nine *assignment operators.* A *simple assignment* is one made using the simple assignment operator `=`. A *compound assignment* uses one of the eight other operators, which compound a binary operator with an assignment.

An assignment operator has two operand expressions. The first is known as the *left-hand side* and has a form restricted to representing a local name, an output parameter or a (possibly indexed) attribute. The left-hand side denotes the target to be assigned by the assignment expression. The second operand expression is known as the *right-hand side.* The right-hand side expression evaluates to the value or values that are assigned to the target designated by the left-hand side.

All assignment operators are syntactically right-associative (they group right-to-left). Thus, an expression of the form `a=b=c` is equivalent to `a=(b=c)`, which assigns the value of c to b and then assigns the value of `b` to `a`.

Examples

```
customer = new Customer()                // Local name assignment
customer[i] = new Customer()             // Indexed local name assignment
reply = this.createReply(request,result) // Output parameter assignment
customer.email = checkout.customerEmail  // Attribute assignment
customer.address[i] = newAddress         // Indexed attribute assignment
x += 4                                   // Compound assignment
filename += ".doc"                       // Compound assignment
```

Syntax

```
AssignmentExpression(e: AssignmentExpression)
      = LeftHandSide(e.leftHandSide) AssignmentOperator(e.operator)
        Expression(e.rightHandSide)
LeftHandSide(lhs: LeftHandSide)
      = NameLeftHandSide(lhs) [ Index(lhs.index) ]
      | FeatureLeftHandSide(lhs) [ Index(lhs.index) ]
      | "(" LeftHandSide(lhs) ")"
NameLeftHandSide(lhs: NameLeftHandSide)
      = PotentiallyAmbiguousQualifiedName(lhs.target)
FeatureLeftHandSide(lhs: FeatureLeftHandSide)
      = FeatureReference(lhs.feature)
AssignmentOperator(op: String)
      = "="(op)  | "+="(op) | "-="(op) | "*="(op) | "%="(op) | "/="(op) |
        "&="(op) | "|="(op) | "^="(op) | "<<="(op) | ">>="(op) | ">>>="(op)
```

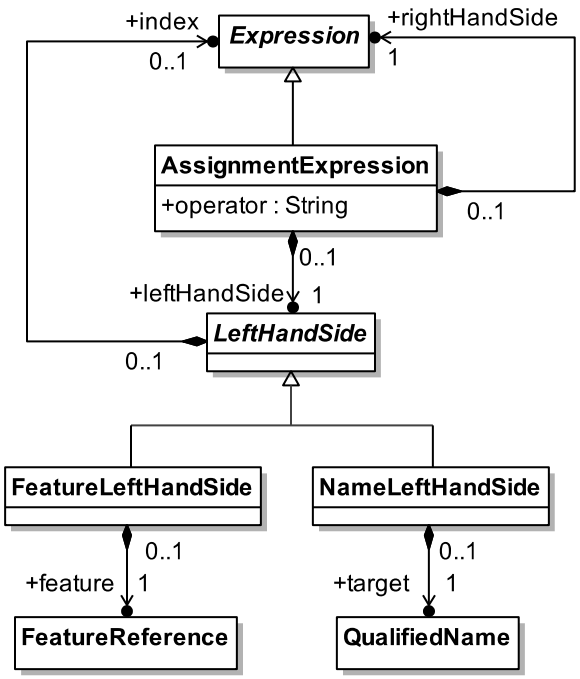

Figure 8.41 Abstract Syntax for Assignment Expressions

Cross References

1. Expression see 8.1
2. QualifiedName see 8.2
3. FeatureReference see 8.3.6
4. Index see 8.3.13

NOTE. See 8.2 for rules on the disambiguation of a qualified name with the dot notation versus a feature reference used as a left-hand side.

Semantics

Left-Hand Side

The left-hand side of an assignment expression may be one of the following:

- *Local name.* The assigned source of the local name before the assignment expression must not be a loop variable definition (see 9.12), a `@parallel` annotation (see 9.12) or a sequence expansion expression (see 8.3.19).
- *Output parameter name.* The named parameter must have mode `out` or `inout`. A parameter name may be qualified with the name of the behavior or operation that owns it, though this is not required. The identified parameter must not be a template. The assignment expression must appear within the definition of the behavior that owns the parameter or the method of the operation that owns the parameter.
- *Property reference.* A property reference is a feature reference that names a property of the type of its primary expression. As for a property access expression (see 8.3.6), the identified property may be either a structural feature or an association end, but must not be a template. The primary expression of the property reference must have a multiplicity upper bound of 1.

A left-hand side may also include an index. An index expression must have a type that conforms to type `Integer` and a multiplicity upper bound no greater than 1. An index is only allowed on a local or parameter name if the name has been previously assigned and on a property reference if the named property is ordered.

The type of the left-hand side is determined as given below:

- *Local Name.* Determined by its first assignment. If this is the first assignment of the local name, then the type of the left-hand side is the type of the right-hand side expression.
- *Parameter Name.* As declared for the named parameter.
- *Property Reference.* As given for the named property.

If the left-hand side has an index, then the multiplicity of the left-hand side is `[0..1]`. Otherwise, the multiplicity of the left-hand side is determined as given for its type above. If this is the first assignment of the local name, then the multiplicity lower bound for the new local name is 0 if the multiplicity lower bound of the right-hand side is 0 and 1 otherwise; the multiplicity upper bound of the new local name is 1 if the multiplicity upper bound of the right-hand side is 1 and `*` otherwise..

Assignability

The right-hand side of an assignment must be *assignable* to the left-hand side. In general, this means that the right-hand side is *statically compatible* in type and multiplicity with the left-hand side, either directly or after the application of a *conversion.*

A right-hand side is assignable to a left-hand side if any of the following conditions hold:

1. *Conformance.* The left-hand is untyped or the right-hand side has a type that conforms to the type of the left-hand side (see 8.2 for the definition of type conformance). If the multiplicity upper bound of the left-hand side is less than or equal to 1, then the multiplicity upper bound of the right-hand side cannot be greater than that of the left-hand side. If the multiplicity lower bound of the left-hand side is greater than 0, and the left-hand side is *not* a local name, then the multiplicity lower bound of the right-hand side cannot be 0.

2. *Null Conversion.* The right-hand side is untyped with a multiplicity of `[0..0]` (i.e., guaranteed to be `null`) and the left-hand side has a multiplicity lower bound of 0 (regardless of type).
3. *Collection Conversion.* The type of the right-hand side is a collection class (see 11.7), the right-hand side has a multiplicity upper bound of 1 and the type and multiplicity of the result of applying the `toSequence` operation to the right-hand side would be assignable to the left-hand side. The assigned value is the result of implicitly calling the `toSequence` operation on the result of evaluating the right-hand side expression.
4. *Bit String Conversion.* The type of the right-hand side conforms to `Integer`, and the type of the left-hand side is `BitString`. If the multiplicity upper bound of the left-hand side is less than or equal to 1, then the multiplicity upper bound of the right-hand side cannot be greater than that of the left-hand side. The assigned value is the result of implicitly invoking the standard library `BitStringFunctions::ToBitString` function (see 11.4.6) on each of the values in the result of evaluating the right-hand side expression. Note that *both* collection conversion and bit string conversion may apply. In this case, bit string conversion is applied after collection conversion.
5. *Real Conversion.* The type of the right-hand side conforms to `Integer`, and the type of the left-hand side conforms to `Real`. If the multiplicity upper bound of the left-hand side is less than or equal to 1, then the multiplicity upper bound of the right-hand side cannot be greater than that of the left-hand side. The assigned value is the result of implicitly invoking the standard `IntegerFunctions::ToReal` function (see 11.4.2) on each of the values in the result of evaluating the right-hand side expression. Note that *both* collection conversion and real conversion may apply. In this case, real conversion is applied after collection conversion.

The concept of assignability is defined here for assignments, but it can actually be applied between any two typed elements with multiplicity. It is used in this way to define the required compatibility between formal parameters and arguments in the tuples of an invocation expression (see 8.3.9). When assignability is used in this general way, the term “left-hand side” used here in its definition should be read as “the target element” and the term “right-hand side” should be read as “the source element”.

Simple Assignment

A simple assignment has the form `lhs = expr`. The right-hand side expression of a simple assignment must be *assignable* to the left-hand side, as defined above.

When a simple assignment expression is evaluated, the right-hand side expression is evaluated first. If the left-hand side contains an index expression, this is evaluated *after* the right-hand side expression. The result of the right-hand side expression is then assigned to the left-hand side, as described below (possibly after conversion as discussed above).

If the left-hand side does *not* have an index expression, then the assignment proceeds as follows:

- *Local name.* If the local name already exists, then it is assigned a new value, with its static assigned source considered to have a the type of the right-hand side expression as its known subtype (see also the discussion of local name multiplicity and type adjustment in 8.7). Otherwise, the assignment expression acts as the definition of a new local name with the type of the right-hand side expression. In either case, the multiplicity for the local name after the assignment is determined based on the multiplicity of the right-hand side expression, as described above for when the local name is being newly defined.

NOTE. It is *not* required to declare a local name before its first assignment. A first assignment can be used to implicitly define a new local name. However, if a local name is defined using a local name declaration statement (see 9.6), that is considered to be its first assignment.

- *Output parameter name.* A new value is assigned to the named parameter, with its static assigned source considered to have a the type of the right-hand side expression as its known subtype (see also the discussion of local name multiplicity and type adjustment in 8.7). Note that any previously assigned value is effectively overwritten. That is, at the completion of the execution of a behavior, output parameters are given their last assigned value. The multiplicity of the new assigned source for the parameter is determined based on the multiplicity of the right-hand side expression in the same way as for a local name (note that this may be different than the multiplicity of the left-hand side containing the parameter, which will be the same as the declared multiplicity of the parameter).
- *Property reference.* A new value is assigned to the named property of the referenced object or structured data value.

  If the property reference has a primary expression that is a local name or parameter name and has a type that is a structured data type, then the assignment to the property reference effectively assigns a new data value to that local or parameter name, with the given property updated.

If the left-hand side includes an index, then only the value at the index position is overwritten by the right-hand side value. However, if the right-hand side value is an empty collection, then the former value at the index position is removed without being replaced by a new value. Indexing is from 1, unless the assignment expression is contained, directly or indirectly, within a statement to which the annotation `@indexFrom0` applies (see 9.2), in which case indexing is from 0.

NOTE. Since `null` represents the empty collection, not a value itself, an expression of the form `x.a[2] = null` will *remove* the second value of the collection `x.a`, *not* assign some "`null`" value to it.

A simple assignment expression has the type and multiplicity of its right-hand side expression.

As noted earlier for a property reference left-hand side, an assignment expression may also be used to update a binary association in which an instance participates via reference to the opposite association end. In this case, the effect of the assignment is equivalent to an appropriate link operation (see 8.3.13).

As an example of an association end update, consider the following association (represented in Alf notation—see 10.4.5).

```
assoc Owns {
  owner: Person;
  house: House[*];
}
```

If the association `Owns` is in the current scope (that is, visible without qualification), and `newHouse` is a `House`, then the expression

```
newHouse.owner = jack;
```

is equivalent to the link operation

```
Owns.createLink(owner => jack, house => newHouse)
```

The result value of an assignment expression is the value of the right-hand side. Thus, the expression

```
WriteLine(a = "x")
```

assigns 3 to `a` and then writes the value “x”. Note that this should not be confused with the named tuple notation, such as

```
WriteLine(value => "x")
```

which uses the “=>” symbol, rather than “=”.

Compound Assignment

A *compound assignment* has the form `lhs op= expr`, where `op` is any arithmetic (sees 8.6.1) or logical (see 8.6.7) operator. It is equivalent to `lhs = lhs op expr` (except that, if `lhs` contains an index expression, it is only evaluated once).

In a compound assignment expression, if the left-hand side is a name (qualified or unqualified), it must also satisfy the semantics of a name expression (see 8.3.3). If the left-hand side is a feature reference, then it must also satisfy the static semantics of a property access expression (see 8.3.6). If the left-hand side has an index, then, in addition to the requirements for its name or feature reference, the left-hand side overall must also satisfy the semantics of a sequence access expression (see 8.3.16).

Both the left-hand side and the right-hand side must have multiplicity upper bounds of 1 and must have the types that are consistent with the arithmetic or logical operator used in the compound assignment operator (sees 8.6.1 or 8.6.7, respectively), that is, such that the equivalent expression `lhs op expr` is legal.

9 Statements

9.1 Overview

Statements are segments of behavior that are executed for their effect and do not have values. They are the primary units of sequencing and control in the Alf representation of behavior. This clause defines the kinds of statement allowed in Alf.

The *full conformance* level includes all kinds of statements specified in this clause. However, the *minimum conformance* level only requires a subset of the full statement syntax. Therefore, in each of the concrete syntax grammar productions given in the subclauses of this clause, some portion of the production may be italicized. Only the italicized portions apply at the minimum conformance level. Unitalicized portions may be ignored for minimum conformance. (See also 2.2 on the definition of syntactic conformance.)

A *statement sequence* is an Alf text consisting of a list of statements juxtaposed in a linear order. Such statement sequences may be attached to UML models in order to specify behaviors. A *block* is a delineation of a statement sequence for use as a component of a larger syntactic construct.

Examples

```
{
  'activity' = (Activity)(this.types[1]);
  group = new ActivityNodeActivationGroup();
  group.activityExecution = this;
  this.activationGroup = group;
  group.activation('activity'.node, 'activity'.edge);
}
```

Syntax

```
Block(b: Block)
      = "{" StatementSequence(b) "}"
StatementSequence(b: Block)
      = { DocumentedStatement(b.statement) }
DocumentedStatement(s: Statement)
      = [ DocumentationComment(s.documentation) ] Statement(s)
Statement(s: Statement)
      = AnnotatedStatement(s)
      | InLineStatement(s)
      | BlockStatement(s)
      | EmptyStatement(s)
      | LocalNameDeclarationStatement(s)
      | ExpressionStatement(s)
      | IfStatement(s)
      | SwitchStatement(s)
      | WhileStatement(s)
      | DoStatement(s)
      | ForStatement(s)
      | BreakStatement(s)
      | ReturnStatement(s)
      | AcceptStatement(s)
      | ClassifyStatement(s)
```

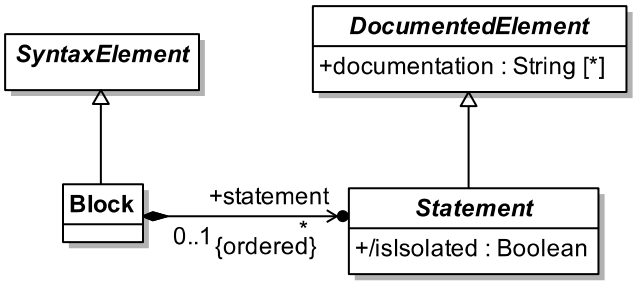

Figure 9.1 Base Abstract Syntax for Statements and Blocks

Cross References

1. SyntaxElement see 6.6
2. DocumentedElement see 6.6
3. DocumentationComment see 7.5.3
4. AnnotatedStatement see 9.2
5. InLineStatement see 9.3
6. BlockStatement see 9.4
7. EmptyStatement see 9.5
8. LocalNameDeclarationStatement see 9.6
9. ExpressionStatement see 9.7
10. IfStatement see 9.8
11. SwitchStatement see 9.9
12. WhileStatement see 9.10
13. DoStatement see 9.11
14. ForStatement see 9.12
15. BreakStatement see 9.13
16. ReturnStatement see 9.14
17. AcceptStatement see 9.15
18. ClassifyStatement see 9.16

Semantics

Unless otherwise annotated (see 9.3), all the statements in a statement sequence are executed sequentially in order. See also the discussion for each kind of statement in the following subclauses.

Integration with UML

An Alf *statement sequence* can be inserted into a UML model using an *opaque behavior* (UML Superstructure, 13.3.20) in which the unprocessed text of the Alf statement sequence is used as the body of the opaque behavior and the corresponding language string is `"Alf"`. Opaque behaviors may be used in a UML model any place that a behavior is allowed.

An Alf statement sequence can also be inserted into a UML activity using an *opaque action* (UML Superstructure, 11.3.26) in which the unprocessed text of the Alf statement sequence is used as the body of the opaque action and the corresponding language string is `"Alf"`. The input and output pins of such an action must all be named. The input pins are considered to be the *assigned sources* (see 8.1) for their names within the statement sequence. The names of the output pins may be assigned within the statement sequence, and their assigned sources at the end of the statement sequence provide the values for the corresponding output pins. The names of other visible model elements (qualified as necessary) may also be used as usual within the statement sequence (see 8.3.3 on name expressions).

The execution semantics of Alf statements are given formally by the mapping to UML activity graphs given in the following subclauses. An Alf statement sequence can therefore always be *compiled* to a part or all of a UML activity model (which does not necessarily need to be the same as the formal mapping, but must have an equivalent effect to it—see 2.3).

Indeed, the semantics of an opaque behavior that only includes unprocessed Alf text in its body may be considered equivalent to an activity with the same parameters as the opaque behavior, containing the activity nodes and edges mapped from the body of the opaque behavior. The semantics of an opaque action that only includes unprocessed Alf text in its body may be considered equivalent to a structured activity node with the same input and output pins as the opaque action, containing the activity nodes and edges mapped from the body of the opaque behavior.

Local Names

Local names (see 8.1) are used in Alf to denote intermediate values in computations within a statement sequence. The scope of such a local name is from the point at which it is defined lexically to the end of the outermost enclosing statement sequence, excluding alternate clauses of the same `if`, `switch` or `accept` statement (see 9.8, 9.9 or 9.15, respectively). Alf does *not* provide hierarchical scoping of local names defined in nested blocks.

The assignments of and references to local names in an Alf input text need to be mapped to appropriate object flow edges from the mapping of the appropriate assignment to the mapping of the reference that requires that assigned value. The *assigned source* for a local name is the Alf element that, when executed, will provide the actual *assigned value* for that local name. If the assigned source for a local name is known, then a reference to the assigned value of that local name can be mapped to an object flow from the mapping of the assigned source.

Since local names can be reassigned in Alf (see 8.8 on assignment expressions), the assigned source for any given local name can be different at different points in a statement sequence. In order to carry out the above mapping, it must always be possible to determine *at most one* assigned source for any local name at any point in the text. The general rule is that the latest assignment “lexically previous” to a reference to a local name is used as the assigned source for that reference. However, some care must be given to carefully defining this term when assignments are allowed within control structures such as conditional and looping statements.

The static semantic rules for local names provide the necessary formalization for determining assigned sources. As with other static semantic rules, these rules are applied by traversing the abstract syntax tree of a statement sequence. At each point in this traversal, the rules determine the set of local names with assigned sources. A name that has no assigned source is known as an *unassigned* name.

When the analysis reaches a specific node in the abstract syntax tree representing a certain kind of Alf construct, a local name with an assigned source is said to have that source *before* the construct in question. If the name is unassigned, then it is said to be unassigned *before* the construct. The assignment

rules then determine what the assigned source for the name is *after* that construct, continuing the traversal to the next node in the parse tree. The rules may also place restrictions on what assignments are allowed, in order that the analysis may be carried out.

NOTE. The assignment rules for statements often refer to the rules for assigned values before and after the evaluation of expressions within the statement. These rules are given for each kind of expression in the subclauses of Clause 8.

The assigned source for a name before the first statement of a statement sequence is the same as the assigned source before the statement sequence. The assigned source for a name before each statement other than the first is the assigned source after the previous statement (as determined by the rules of the following subclauses for each kind of statement). The assigned source for a name after a statement sequence is the same as the assigned source after the last statement of the sequence.

Note that a block in Alf is basically just a syntactic delineation of a sequence of statements and does not introduce new semantics as such. In particular, a block does *not* introduce a lexically nested scope for local names. That is, local names with assigned sources lexically previous to and visible from the statement sequence of a block cannot be redefined in the block (as with a local name declaration statement—see 9.6), though they can generally be reassigned unless there is some specific rule otherwise (such as the special rules for reassignment of local names in a parallel for statement—see 9.12).

9.2 Annotated Statements

Certain statements may have *annotations* that effect the execution of the statement. An annotation has the form of an identifier preceded by a "`@`" character. Note that the identifiers for annotations are *not* reserved words, but only a limited set of predefined annotation identifiers can be used (see Table 9.1).

A set of annotations for a statement are listed on one or more lines preceding the statement. Each of these lines has a similar form to an end-of-line lexical comment (see 7.5). This syntax reflects the fact that an annotation is essentially a directive used in the mapping of the annotated statement, not an executable construct in its own right. An annotation may place restrictions on the allowed form of the statement being annotated, but that statement is still always legal even if the annotation is removed.

Examples

```
//@isolated
{
  temperature = temperatureSensor.read();
  pressure = pressureSensor.read();
  ActuateControl(temperature, pressure);
}
```

Syntax

```
AnnotatedStatement(s: Statement)
      = "//@" Annotation(s.annotation) { "@" Annotation(s.annotation) }
        [ "//" { InputCharacter } ] LineTerminator
       Statement(s)
Annotation(a: Annotation)
      = Identifier(a.identifier)
        [ "(" Name(a.argument) { "," Name(a.argument) ")" ]
```

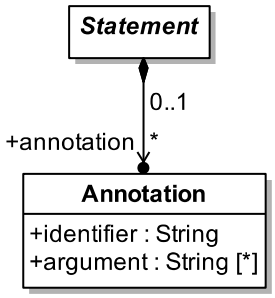

Figure 9.2 Abstract Syntax of Annotations

Cross References

1. LineTerminator see 7.2
2. InputCharacter see 7.2
3. Identifier see 7.6
4. Statement see 9.1

NOTE. The lexical element `LineTerminator` is used as a syntactic non-terminal element in the production for `AnnotatedStatement` above and must *not* be ignored as white space in this case (see also 7.4). Even though the initial "`//@`" token has a form similar to the start of an end-of-line comment (see 7.5), annotations must specifically follow the syntax shown above. However, a list of annotations may be followed by a "`//`", the characters after which are ignored, giving the same effect as having an end-of-line comment at the end of the annotation line.

Semantics

Annotations

Even though syntactically any statement may be annotated, specific annotations may only be used with the specific statements to which they apply. Table 9.1 lists all allowable annotations and the statements to which they apply. The `@isolated`, `@indexFrom0` and `@indexFrom1` annotations apply to all statements (other than empty and `break` statements) and their effect is described below. The effect of each of the other annotations is described in the subclause for the statement (or statements) to which it applies.

Table 9.1 Allowable Annotations

Annotation	Applicable Statement	Allows Arguments?
`isolated`	Any statement other than an empty statement or `break`.	No
`deternmina te`	`if` (see 9.8) `switch` (see 9.9)	No
`assured`	`if` (see 9.8) `switch` (see 9.9)	No
`parallel`	block (see 9.3)	No
	`for` (see 9.12)	Yes

`indexFrom0`	Any statement other than an empty statement or `break`.	No
`indexFrom0`	Any statement other than an empty statement or `break`.	No

Since the syntax allows an annotated statement to itself be annotated, annotations may be spread across multiple lines preceding a single "base" statement. For the purposes of determining applicability, per Table 9.1, of a further annotation of an annotated statement, the annotated statement is to be considered to be of the same kind as its base statement. However, no annotation may be applied more than once to the same base statement.

Arguments

The syntax for annotations allows for an optional list of names to be given as arguments of the annotation. Currently, only the `@parallel` annotation on a `for` statement allows such argument (see 9.12 for rules related to names used as such arguments).

Isolation

The annotation `@isolated` may be used with any statement other than the empty or break statements, indicating that the statement is executed in *isolation.* That is, during the execution of the statement, no object accessed as part of the execution of the statement or as the result of a synchronous invocation from the statement may be modified by any action that is not executed as part of the statement or as the result of a synchronous invocation from the statement.

The semantics of isolation is discussed in 8.5.4.1 of the fUML specification.

NOTE. The unary operator `$` can also be used to denote isolation at the expression level (see 8.5.6).

Indexing from 0 and 1

The annotation `@indexFrom0` may be used with any statement other than the empty or `break` statements to indicate that, in expressions within that statement, indexing of the positions of elements in a sequence will start from 0 for the first element, rather than the default, which is indexing from 1. This annotation effects behavior invocation expressions and sequence operation expressions for certain library functions (see 8.3.9 and 8.3.17), link operation expressions with indexed arguments (see 8.3.17), sequence access expressions (see 8.3.16) and indexed left-hand sides (see 8.8).

The scope of an `@indexFrom0` annotation includes any expressions contained directly in the annotated statement, as well as any statements nested, directly or indirectly, in the annotated statement, unless a nested statement has the annotation `@indexFrom1` applied. The annotation `@indexFrom1` has the converse effect to `@indexFrom0`, restoring the default of indexing from 1 for all contained expressions and nested statements, other than nested statements that have the annotation `@indexFrom0` explicitly applied.

NOTE. In UML, the indexing of the positions of elements in a sequence always starts with 1 for the first element, so this is also the default for Alf. However, in other languages with a syntax similar to Alf's, indexing is usually from 0 for the first element. The `@indexFrom0` annotation is provided to ease the adaption of existing code from such languages into Alf text. However, use of indexing from 1 is generally preferred, since this is consistent with standard UML modeling semantics.

9.3 In-line Statements

An *in-line statement* allows code in a language other than Alf to be inserted in-line as an Alf statement. The actual interpretation and execution of such inline code is implementation dependent. Typically, such code would be passed directly to a target implementation platform, but the details are not defined in the Alf standard. While syntactically a statement, an in-line statement has a form similar to an in-line lexical comment, to indicate that the in-line code is not included in normal Alf language processing.

The language used for the in-line code is identified by a name. There is no standard list of language names, but the following names are recommended to promote potential interoperability for commonly embedded language fragments.

- `Java`
- `C`
- `'C++'`
- `'C#'`

Example

```
/*@inline('C++')  // Native code
  *data = this;
  controller->initiate();
*/
```

Syntax

```
InLineStatement(s: InLineStatement)
      = InLineHeader(s) CommentText(s.code) "*/"
InLineHeader(s: InLineStatement)
      = "/*@" "inline" "(" Name(s.language) ")"
        [ "//" { InputCharacter } ] LineTerminator
```

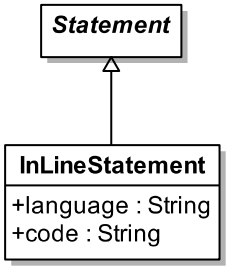

Figure 9.3 Abstract Syntax of In-Line Statements

Cross References

1. LineTerminator see 7.2
2. InputCharacter see 7.2
3. CommentText see 7.5.2
4. Name see 7.6

NOTE. The lexical element `LineTerminator` is used as a syntactic non-terminal element in the production for `InLineHeader` above and must *not* be ignored as white space in this case (see also 7.4). Even though the initial "`/*@`" token has a form similar to the start of an in-line comment (see 7.5), an in-line statement header must specifically follow the syntax shown above. However, the `inline` annotation

may be followed by a "//", the characters after which are ignored to the end of line, giving the same effect as having an end-of-line comment on the header line. Comments may also be used in the body code of an in-line statement, as allowed by the specific language in which that code is written. However, since the comment delimiter "*/" is used to end an in-line statement, comment syntax using this delimiter cannot be used within the in-line statement.

Semantics

The execution semantics for an in-line statement are implementation specific.

Any relationship of code within an in-line statement to named elements outside of the in-line statement is also implementation specific.

9.4 Block Statements

A block (see 9.1) may itself be executed as a statement.

Examples

```
{ index = this.getIndex(); this.list[index] = this.update(index); }
```

The example below represents the activity graph shown in Figure 9.4.

```
//@parallel
{
  'activity' = (Activity)(this.types[1]);            // Segment 1

  {                                                  // Segment 2
    group = new ActivityNodeActivationGroup();
    group.activityExecution = this;
  }

  {                                                  // Segment 3
    this.activationGroup = group;
    group.activation('activity'.node, 'activity'.edge);
  }
}
```

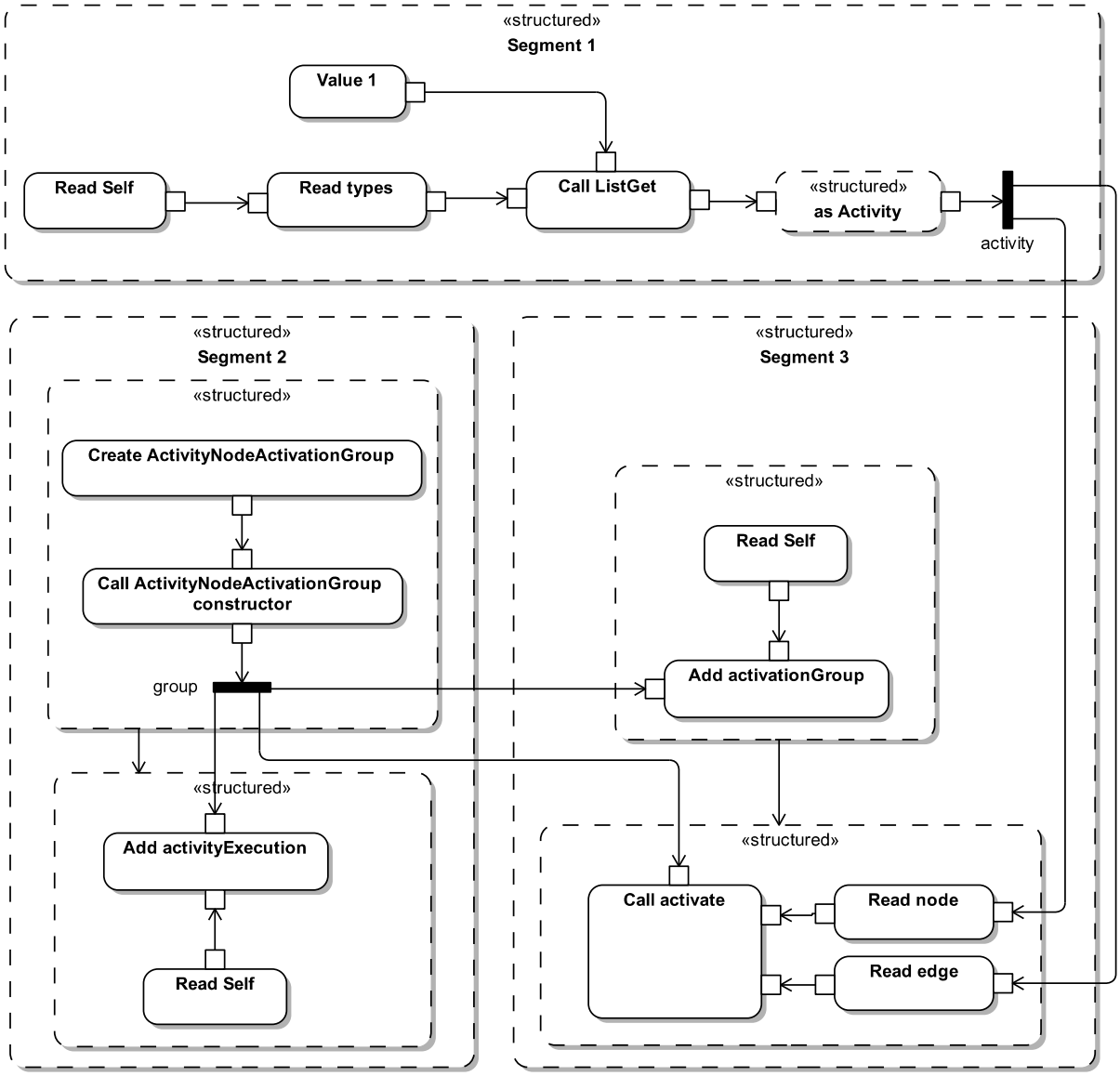

Figure 9.4 Sample Activity Diagram

Syntax

```
BlockStatement(s: Statement)
      = Block(s.block)
```

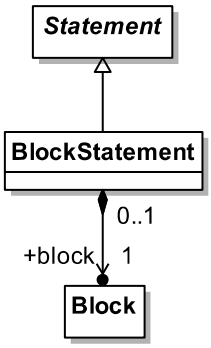

Figure 9.5 Abstract Syntax for Block Statements

Cross References

1. Statement see 9.1
2. Block see 9.1

Semantics

Sequential Execution

The execution of a block statement consists simply in the execution of the block (see 9.1). Unless the block statement has a `@parallel` annotation (see below), this means that the statements in the block are executed sequentially in order.

Local names assigned before a block statement may be reassigned within the constituent block of the statement. Further, new local names may be defined within the block. Such new local names are available outside the block after completion of execution of the block statement, with the values they held at the completion of execution of the block. Thus, blocks do *not* introduce lexically nested scopes for defining local names in Alf.

Parallel Execution

A block statement may have a `@parallel` annotation. In this case, the statements in the block are all executed concurrently. All the statements of a parallel block are enabled to start executing when the block is executed and the block does not complete execution until all statements complete their execution (that is, there is an implicit *join* of the concurrent executions of the statements).

If a block statement has a `@parallel` annotation, any name assigned in one statement of the block may not be further assigned in any subsequent statement in the same block.

NOTE. The above rule allows a name assigned in one statement in a parallel block to be used in a subsequent statement, but not reassigned. This allows data dependencies between otherwise parallel executions of these statements. However, such dependencies are somewhat restricted, since a newly defined name is still only visible in lexically following statements, as for a sequential block. This restriction prevents the possibility of a block statement being invalidated simply by removing a `@parallel` annotation from it.

For example, in the block

```
//@parallel
{
  a = F(1);
```

```
  b = G(2);
}
```

the invocations of the behaviors `F` and `G` will execute concurrently with no dependencies between them. The block as a whole completes execution only after *both* invocations are complete.

However, there may be data dependencies between the statements in a parallel block. For example, the statements in the block

```
//@parallel
{
   a = F(1);
   b = G(a);
}
```

will execute sequentially, despite the `@parallel` annotation, because the invocation of `G` cannot proceed until a value is available for `a`.

In order to prevent data conflicts, it is illegal to assign the same name in more than one statement in a parallel block. Thus, the block

```
//@parallel
{
  a = 1;
  a = F(a); // Illegal!
  b = G(2);
}
```

is illegal, since the name `a` is reassigned in the second statement. However, the following block *is* legal:

```
//@parallel
{
  {
    a = 1;
    a = F(a); // Legal
  }
  b = G(2);
}
```

This is because the block now consists of just two statements, the first of which happens to be a (sequential) block statement. Only the first of these actually assigns a value to the name `a`. The value of `a` after the first statement in the parallel block, and hence after the parallel block as a whole, is the value given to it (sequentially) by the second assignment.

9.5 Empty Statements

An empty statement does nothing.

Syntax

```
EmptyStatement(s: EmptyStatement)
      = ";"
```

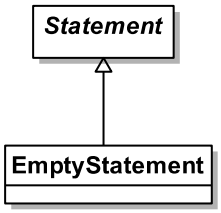

Figure 9.6 Abstract Syntax of Empty Statements

Cross References

1. Statement see 9.1

Semantics

Any empty statement has no effect when it is executed.

9.6 Local Name Declaration Statements

A *local name declaration* is used to define a local name, declare its type and provide an initial value for it. The local name declaration statement has two forms. In the first form, the local name being defined precedes the type name:

```
let interest : CurrencyAmount = this.principal * this.rate * period;
```

In the second form, the type name precedes the local name:

```
CurrencyAmount interest = this.principal * this.rate * period;
```

The two forms are completely equivalent. In both cases, the local name is assigned the result of evaluating the expression.

NOTE. The first form of local name declaration is consistent with the usual UML notation for declaring the type of a name. The second form is consistent with the form of declarations in many common programming languages and also emphasizes the essential semantics of the statement as an assignment with an added type constraint.

Examples

```
let currentOffer : Offer = this.offers[1];

let origin : Point = new(0,0);

CurrencyAmount interest = this.principal * this.rate * period;

let inactiveMembers : Member[] = members -> select member (!member.active);

RealProperty[] realProperties = (RealProperty)assets;

IntegerList list = {1, 2, 3};

Set<RealProperty> fixedAssets = new { land, home, condominium };
```

Syntax

```
LocalNameDeclarationStatement(s: LocalNameDeclarationStatement)
      = NameDeclaration(s) "=" InitializationExpression(s.expression) ";"
InitializationExpression(e: Expression)
      = Expression(e)
      | SequenceInitializationExpression(e)
      | InstanceInitializationExpression(e)
```

```
InstanceInitializationExpression(e: InstanceCreationExpression)
      = "new" Tuple(e.tuple)
NameDeclaration(s: LocalNameDeclarationStatement)
      = "let" Name(s.name) ":" TypeName(s.typeName)
        [ MultiplicityIndicator (s.hasMultiplicity=true) ]
      | TypeName(s.typeName)
        [ MultiplicityIndicator (s.hasMultiplicity=true) ] Name(s.name)
```

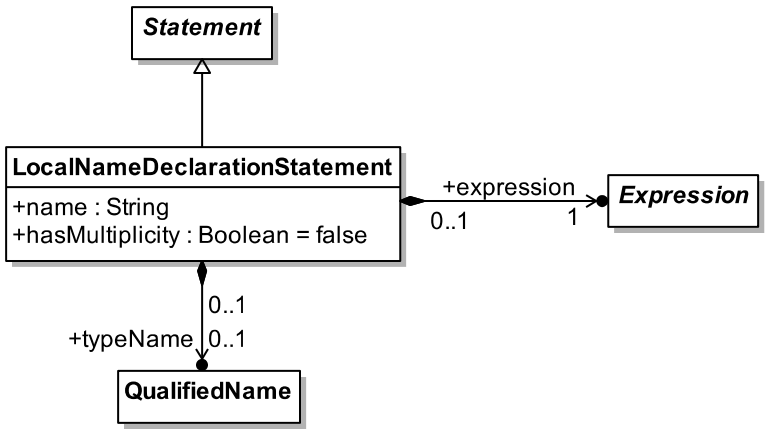

Figure 9.7 Abstract Syntax of Local Name Declaration Statements

Cross References

1.	Name	see 7.6
2.	Expression	see 8.1
3.	QualifiedName	see 8.2
4.	TypeName	see 8.2
5.	InstanceCreationExpression	see 8.3.12
6.	SequenceInitializationExpression	see 8.3.15
7.	MultiplicityIndicator	see 8.3.15
8.	Statement	see 9.1

Semantics

The local name in a local name declaration statement must be unassigned before the statement and before the expression in the statement. It must remain unassigned after the expression. The assigned source for the name after the local name declaration statement is the statement itself.

Typing

The new local name has the type given by the type name in the statement (see 8.2 on type names). The multiplicity lower bound of the local name is 0 if the assigned expression has a lower bound of 0, otherwise it is 1. If the multiplicity indicator "`[]`" is specified with the type, then the multiplicity upper bound of the local name is `*`, otherwise it is 1 and the multiplicity upper bound of the assigned expression must not be greater than 1.

Alf does not require that a local name be explicitly declared before its first use. If not explicitly declared, the type of the local name is determined by its first assignment (see 8.8). However, if the type is

explicitly declared, it may be more general than the type of the initially assigned expression, allowing a wider range of values in later assignments.

For example, the following is legal:

```
let v: Vehicle = new Car();
v = new Truck();
```

presuming that `Car` and `Truck` are both subclasses of `Vehicle`. However, the following would *not* be legal:

```
v = new Car();
v = new Truck();  // Type error!
```

because the initial assignment would determine the type of v as being `Car`, which is not compatible with `Truck`.

A similar effect to the legal assignment above can be achieved using a type cast (see 8.5.5):

```
v = (Vehicle)new Car();
v = new Truck();  // Legal
```

Even though the initial value assigned to `v` here is a `Car`, the type of the expression "`(Vehicle)new Car()`" is `Vehicle`, due to the cast. Therefore, the initial assignment determines the type of `v` as being `Vehicle`, so the second assignment is legal.

However, there is an important semantic difference. Type casts in Alf filter values that cannot be cast, so that

```
v = (Vehicle)new House();
```

is legal and will assign `null` to `v`. On the other hand, the following is *not* legal:

```
let v: Vehicle = new House();  // Type error!
```

since the class `House` does not statically conform to the declared type `Vehicle`.

Initialization Expressions

Since a local name declaration statement already includes an explicit declaration of the type of a new name, it is possible to use simplified forms for sequence construction expressions (see 8.3.15) and instance creation expressions (see 8.3.12) used as initialization expressions in local name declaration statements. In these simplified forms, the explicit type part usually included in the expressions may be omitted, with the type declared as part of the statement being used instead.

For example, in the statement

```
let fixedAssets: RealProperty[] = { land, home, condominium };
```

the type part normally required for a sequence construction expression has been omitted. The statement is equivalent to one in which the type part for the sequence construction expression is the same as the type declared for the new local name:

```
let fixedAssets: RealProperty[] = RealProperty[]{ land, home, condominium };
```

This simplified form may also be used when initializing collection objects. Thus, the statement

```
let fixedAssets: Set<RealProperty> = { land, home, condominium };
```

is equivalent to

```
let fixedAssets: Set<RealProperty> =
  Set<RealProperty>{ land, home, condominium };
```

Similarly, in the statement

```
let origin : Point = new(0,0);
```

the explicit constructor description has been omitted from the instance creation expression, leaving only the keyword `new` and the tuple. The statement is equivalent to one in which the instance creation expression is explicitly for the type declared for the new local name:

```
let origin : Point = new Point(0,0);
```

NOTE. The initialization expression short hands can also be used in Alf expressions representing default values, either as integrated with a non-Alf representation of a property or parameter (see 8.1) or as the initializer for the definition of an attribute in Alf (see 10.5.2).

9.7 Expression Statements

An *expression statement* simply consists of an expression (see Clause 8) followed by a semicolon.

Examples

```
currentOffer = this.offers[1];
```

```
monitor.SignalAlarm(sensorId);
```

```
this.interest = this.principal * this.rate * period;
```

Syntax

```
ExpressionStatement(s: ExpressionStatement)
      = Expression(s.expression) ";"
```

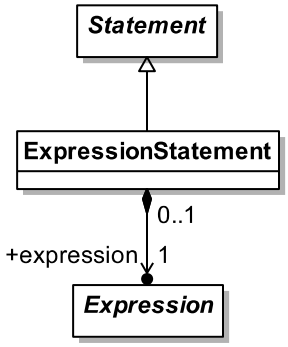

Figure 9.8 Abstract Syntax of Expression Statements

Cross References

1. Expression see 8.1
2. Statement see 9.1

Semantics

An expression statement is executed by evaluating the expression (see Clause 8). If the expression produces one or more values, these are discarded.

The values assigned to any local names within an expression statement (e.g., via an assignment expression—see 8.8) may be accessed after the execution of the statement by using the local names in name expressions (see 8.3.3).

9.8 `if` Statements

An `if` *statement* allows for the conditional execution of one of a set of blocks of statements. The conditional blocks are organized into sequential sets of concurrent clauses. Each clause in a concurrent set includes a conditional expression (which must be of a type that conforms to type `Boolean`) and a block to execute if that condition is true. An `if` statement may also optionally have a final clause with a block to execute if no other conditions are true.

NOTE. The notation for concurrent clauses is not available at the minimum conformance level (see 2.2).

Examples

```
if (reading > threshold) {
   monitor.raiseAlarm(sensorId);
}
```

```
//@determinate @assured
if (reading <= safeLimit) {
  condition = normal; }
or if (reading > safeLimit && reading <= criticalLimit) {
  condition = alert; }
or if (reading > criticalLimit) {
  condition = critical; }
```

Syntax

```
IfStatement(s: IfStatement
      = "if" SequentialClauses(s) [ FinalClause(s.finalClause) ]
SequentialClauses(s: IfStatement)
      = ConcurrentClauses(s.nonFinalClauses)
        { "else" "if" ConcurrentClauses(s.nonFinalClauses) }
ConcurrentClauses(c: ConcurrentClauses)
      = NonFinalClause(c.clause) { "or" "if" NonFinalClause(c.clause) }
NonFinalClause(c: NonFinalClause)
      = "(" Expression(c.expression) ")" Block(c.block)
FinalClause(b: Block)
      = "else" Block(b)
```

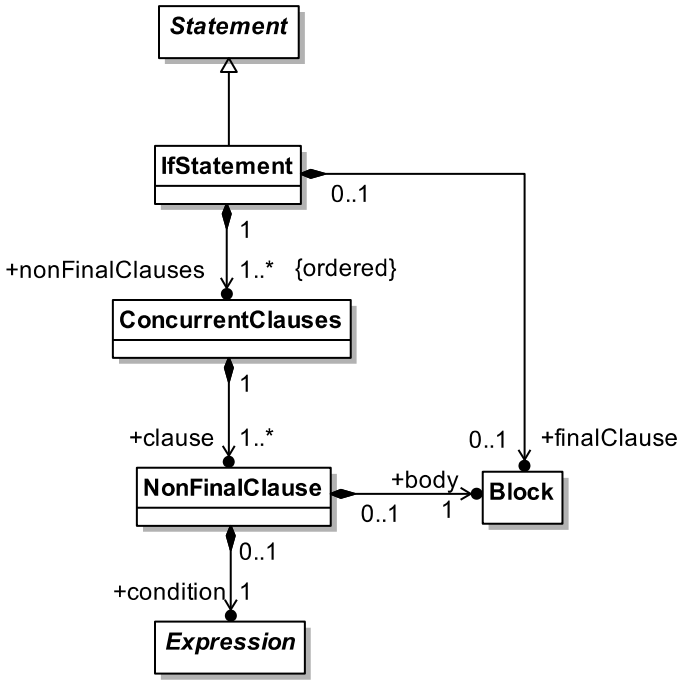

Figure 9.9 Abstract Syntax of `if` Statements

Cross References

1. Expression see 8.1
2. Statement see 9.1
3. Block see 9.1

Semantics

Each set of concurrent clauses is executed in sequence until a condition evaluates to true. Within each set, all the conditional expressions are evaluated in parallel. If any condition evaluates to true, then the associated block may be executed. If more than one condition evaluates to true, then one associated block is chosen non-deterministically to execute. If no conditions evaluate to true, execution proceeds with the next set of concurrent clauses.

All the condition expressions of an `if` statement must have type `Boolean` and a multiplicity of `[1..1]`.

Sequential Clauses

In its simplest form, an `if` statement has a single condition which determines whether or not a single block is executed. For example, in

```
if (reading > threshold) {
   monitor.raiseAlarm(sensorId);
}
```

the invocation is executed only if the condition `reading > threshold` is true. A final clause may be added, as in

```
if (reading > threshold) {
   monitor.raiseAlarm(sensorId);
}
```

```
else {
   monitor.logReading(sensorId, reading);
}
```

In this case, the `logReading` operation is called if `reading > threshold` is false.

An `if` statement may also have a list of conditional clauses that are tested sequentially. For example,

```
if (reading <= safeLimit) {
  condition = normal;
}
else if (reading <= criticalLimit) {
  condition = alert;
}
else {
  condition = critical;
}
```

Concurrent Clauses

Clauses beginning with `or` instead of `else` are tested concurrently rather than sequentially. For example,

```
if (reading <= safeLimit) {
  condition = normal;
}
or if (reading > safeLimit && reading <= criticalLimit) {
  condition = alert;
}
or if (reading > criticalLimit) {
  condition = critical;
}
```

Note that the second condition has the added test `reading > safeLimit`, since the first condition will no longer be evaluated sequentially before the second. If this addition had not been made, then, if, in fact, `reading <= safeLimit` was true, both of the first two conditions could be true, and either of the associated blocks might actually execute (but only one would).

Sequential and conditional clauses can also be mixed, but sets of contiguous concurrent clauses are always sequenced together. Thus, in

```
if (reading <= safeLimit) {
  condition = normal; }
else if (reading > safeLimit && reading <= criticalLimit} {
  condition = alert; }
or if (reading > criticalLimit && reading < errorLimit) {
  condition = critical; }
else {
  condition = error; }
```

if the first condition is false, then both the next two conditions are evaluated concurrently. If both these conditions are also false, then the final clause is executed.

Annotations

The annotations `@assured` and `@determinate` may be used with an `if` statement (see also 9.2 on annotations). The annotation `@assured` indicates that at least one condition in the `if` statement will always evaluate to true. The annotation `@determinate` indicates that at most one condition will evaluate to true. The annotations may be used together, which indicates that exactly one condition will always evaluate to true.

New local names may not be defined in conditional expressions, since these may not always be evaluated, but existing local names may be reassigned. However, the same name may not be assigned in more than one conditional expression within the same concurrent set of clauses, because these expressions are evaluated concurrently, so assignments of the same name in more than one of them could potentially conflict. Assignments made in the conditional expression of a non-final clause are available in the block of that clause.

New local names may be defined in the clause blocks of an `if` statement. If such a name is defined in more than one clause block, its type after the `if` statement is the effective common ancestor (see the definition in 8.7) of the type of the name in each clause in which it is defined with a multiplicity lower bound that is the minimum of the lower bound for the name in each clause and a multiplicity upper bound that is the maximum of the upper bound for the name in each clause. If the name is not defined in every clause of the `if` statement, then it is considered to have multiplicity lower and upper bounds of 0 for the purposes of the above bounds determination. If the clause of the `if` statement that is executed does not define a name that is defined in a different clause, then that name is defined but empty after the execution of the `if` statement.

The multiplicity and typing of local and parameter names referenced in the block of a clause of an `if` statement are adjusted based on the conditional expression of the clause evaluating to true (see 8.7 on local name multiplicity and type adjustment). In addition, the multiplicity and typing are similarly adjusted based on the conditional expressions of all the preceding clauses in the `if` statement evaluating to false.

9.9 `switch` Statements

A *`switch` statement* executes one of several blocks depending on the value of an expression. The body of the `switch` statement consists of a list of clauses, each of which consists of a set of `case` labels and a block. Each of the `case` labels contains an expression that must evaluate to a single value of a type that conforms to the type of the switch expression (see 8.2 for the definition of type conformance). In addition, a `switch` statement may have a final clause with the label `default`.

Examples

```
switch (month) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    numDays = 31;
  case 4: case 6: case 9: case 11:
    numDays = 30;
  case 2:
    if ( ((year % 4 == 0) && !(year % 100 == 0))
         || (year % 400 == 0) ) {
      numDays = 29;
    }
    else {
      numDays = 28;
    }
  default:
    WriteLine("Invalid month.");
    numDays = 0;
}
```

Syntax

```
SwitchStatement(s: SwitchStatement)
      = "switch" "(" Expression(s.expression) ")"
        "{" { SwitchClause(s.nonDefaultClause) }
        [ SwitchDefaultClause(s.defaultClause) ] "}"
SwitchClause(s: SwitchClause)
      = SwitchCase(s.case) { SwitchCase(s.case) }
        NonEmptyStatementSequence(s.block)
SwitchCase(e: Expression)
       = "case" Expression(e) ":"
SwitchDefaultClause(b: Block)
      = "default" ":" NonEmptyStatementSequence(b)
NonEmptyStatementSequence(b: Block)
      = DocumentedStatement(b.statement) { DocumentedStatement(b.statement) }
```

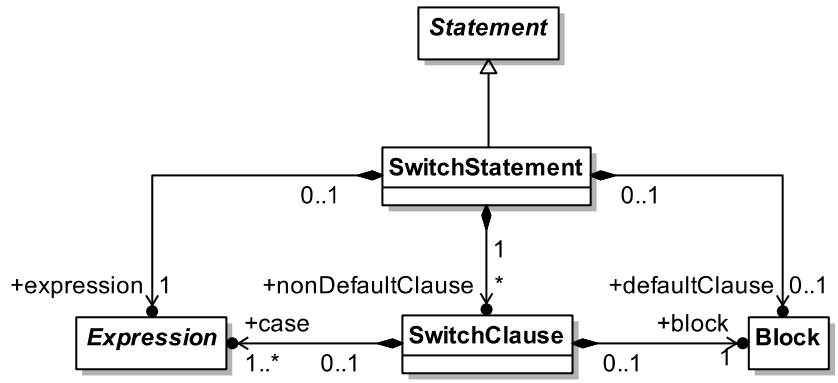

Figure 9.10 Abstract Syntax of `switch` Statements

Cross References

1. Expression see 8.1
2. DocumentedStatement see 9.1

Semantics

Switch Execution

When a `switch` statement is executed, first the switch expression is evaluated and then all the case expressions are evaluated concurrently. The switch expressions and all case expressions must have a multiplicity upper bound no greater than 1. If the result of any case expression equals the result of the switch expression, then the associated statement sequence is enabled for execution. If more than one statement sequence is enabled, then one of them is non-deterministically selected for execution.

If no case expressions have a result equal to the result of the switch expression, and there is a default clause, then the statement sequence associated with the default clause is executed. Since all the case expressions must be checked before a default clause can execute, the execution of the default clause always happens sequentially after the completion (and failure) of all case tests.

If no case expressions have a result equal to the result of the switch expression, and there is not a default clause, then the switch statement has no further effect.

“Equality” of values is evaluated as for an equality operator (see 8.6.6).

NOTE. Case expressions are evaluated concurrently, not sequentially, and execution does *not* "fall through" from one switch clause to the next, as it does in traditional C `switch` statements. This means that it is not necessary to place a `break` statement at the end of a switch clause to avoid execution continuing with the next clause. However, placing a `break` at the end of the clause anyway will not harm the overall execution of the `switch` statement. (See also 9.13 on `break` statements.)

Annotations

The annotations `@assured` and `@determinate` may be used with a `switch` statement (see also 9.2 on annotations). The annotation `@assured` indicates that the result of at least one clause in the switch statement will always be executed. The annotation `@determinate` indicates that at most one clause will execute. The annotations may be used together, which indicates that exactly one clause will always execute.

Names

New local names may not be defined in case expression, but existing local names may be reassigned. However, the same name may not be assigned in more than one case expression because these expressions are evaluated concurrently, so assignments of the same name in more than one of them could potentially conflict. Assignments made in the case expressions of a switch clause are available in the statement sequence of that clause.

New local names may be defined in the statement sequences of a `switch` statement. If such a name is defined in more than one switch clause, its type after the `switch` statement is the effective common ancestor (see the definition in 8.7) of the type of the name in each clause in which it is defined with a multiplicity lower bound that is the minimum of the lower bound for the name in each clause and a multiplicity upper bound that is the maximum of the upper bound for the name in each clause. If the name is not defined in every clause of the `switch` statement, then it is considered to have multiplicity lower and upper bounds of 0 for the purposes of the above bounds determination. If the clause of the `switch` statement that is executed does not define a name that is defined in a different clause, then that name is defined but empty after the execution of the `switch` statement.

9.10 `while` Statements

A `while` statement executes an expression and a block until the value of the expression is false.

Examples

```
while ((last = this.list->size()) > 0) {
  this.list[last].cleanUp();
  this.list->remove(last);
}
```

```
while (file.hasMore()) {
  nextRecord = file.readNext();
  if (nextRecord!=null) {
    checksum = ComputeChecksum(checksum, nextRecord);
  }
}
```

Syntax

```
WhileStatement(s: WhileStatement)
      = "while" "(" Expression(s.condition) ")" Block(s.body)
```

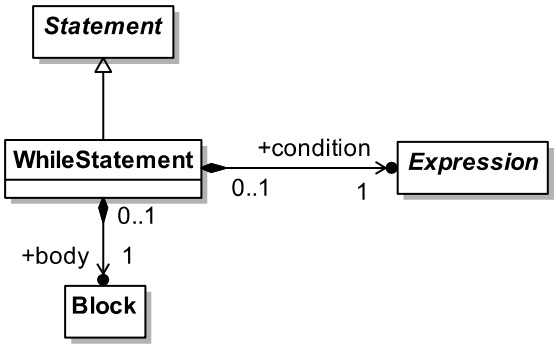

Figure 9.11 Abstract Syntax of `while` Statements

Cross References

1. Expression see 8.1
2. Statement see 9.1
3. Block see 9.1

Semantics

When a `while` statement is executed, its expression is evaluated. If the expression evaluates to false, the execution of the `while` statement is complete. Otherwise, its block is executed, and then the entire `while` statement is executed again, beginning with re-evaluating the expression.

The condition expression of a `while` statement must have a type that conforms to type `Boolean` and a multiplicity of `[1..1]`.

Names

Local names defined within the condition expression of a `while` statement are available within the body block of the `while` statement, with the value as computed on each iteration of the loop. They are also available after completion of execution of the `while` statement, with the values assigned as of the last evaluation of the `while` condition expression.

Local names defined within the body block of a `while` statement may be used within that block. They are also available after the completion of execution of the `while` statement, with values assigned as of the last iteration of the loop. If the loop has no iterations and the body block is not executed, then names defined in that block are defined but empty after the execution of the `while` statement.

Local names defined before a `while` statement may be reassigned within the `while` statement. After completion of execution of the `while` statement, they have the values assigned as of the last evaluation of the `while` statement.

The multiplicity and typing of local and parameter names in the body block of a `while` statement are adjusted based on the condition expression of the `while` statement evaluating to true (see 8.7 on local name multiplicity and type adjustment). The multiplicity and typing of local and parameter names after the `while` statement are adjusted based on the condition expression evaluating to false.

9.11 `do` Statements

The `do` statement executes a block and an expression repeatedly until the value of the expression is false.

Examples

```
do {
  line = file.readNext();
  WriteLine(line);
} while (line != endMarker);
```

```
do {
  reading = sensor.getNextReading();
  Record(reading);
} while (!reading.isFinal());
```

Syntax

```
DoStatement(s: DoStatement)
      = "do" Block(s.body) "while" "(" Expression(s.condition) ")" ";"
```

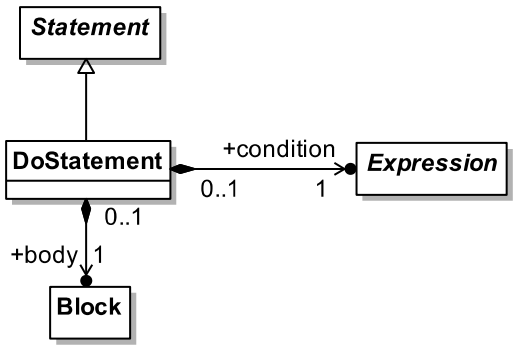

Figure 9.12 Abstract Syntax of do Statements

Cross References

1. Expression see 8.1
2. Statement see 9.1
3. Block see 9.1

Semantics

When a `do` statement is executed, its block is executed first. Then its expression is evaluated. If the result of the expression is true, then the entire `do` statement is executed again. If the result of the expression is false, the execution of the `do` statement is complete.

The condition expression of a `do` statement must have a type that conforms to type `Boolean` and a multiplicity of `[1..1]`.

Names

Local names defined before a `do` statement may be reassigned within the `do` statement and new local names may be defined within the `do` statement, in either the condition expression or the block. Local names defined within the condition expression are not available within the body block, but they are available after completion of the `do` statement, with values assigned as of the last evaluation of the `do` condition expression.

Local names defined within the body block of a `do` statement are available within the condition expression of the `do` statement, with the value as computed on each iteration of the loop. They are also

available after completion of execution of the `do` statement, with the values assigned as of the last evaluation of the `do` condition expression.

Local names defined before a `do` statement may be reassigned within the `do` statement. After completion of execution of the `do` statement, they have the values assigned as of the last evaluation of the `do` statement.

The multiplicity and typing of local and parameter names after a `do` statement are adjusted based on the condition expression of the `do` statement evaluating to false (see 8.7 on local name multiplicity and type adjustment).

9.12 `for` Statements

The `for` statement executes a block repeatedly while assigning a loop variable to successive values of a sequence.

Examples

```
for (member in memberList) {
  names->add(member.name);
  addresses->add(member.address);
}
```

```
for (sensor in sensors) {
  if ((reading = sensor.reading())->isEmpty()) {
    break;
  }
  if (reading > noiseLimit) {
    readings->add(reading);
  }
}
for (i in 1..recordCount) {
  processRecord(i);
}

//@parallel
for (ActivityEdgeInstance outgoingEdge: this.outgoingEdges) {
  outgoingEdge.sendOffer(tokens);
}
```

```
// Fast Fourier Transformation computation
//@parallel(Sn_Even,Sn_Odd)
for (lower in S_Lower, upper in S_Upper, root in V) {
  //@parallel
  {
    Sn_Even -> add(lower+upper);
    Sn_Odd  -> add((lower-upper)*root);
  }
}
```

Syntax

```
ForStatement(s: ForStatement)
       = "for" "(" ForControl(s) ")" Block(s.body)
ForControl(s: ForStatement)
      = LoopVariableDefinition(s.variableDefinition)
        { "," LoopVariableDefinition(s.variableDefinition) }
```

```
LoopVariableDefinition(v: LoopVariableDefinition)
      = Name(v.variable) "in" Expression(v.expression1)
        [ ".." Expression(v.expression2) ]
      | TypeName(v.typeName) Name(v.variable) ":"
        Expression(v.expression1) (typeIsInferred=false)
```

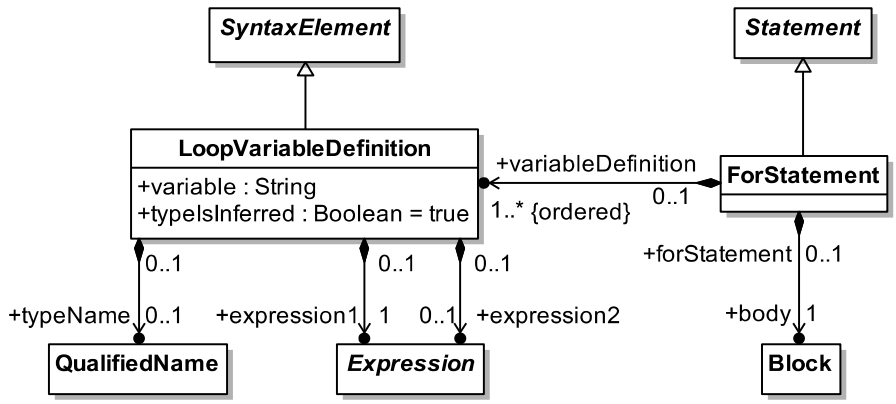

Figure 9.13 Abstract Syntax of `for` Statements

Cross Reference

1. Name see 7.6
2. Expression see 8.1
3. QualifiedName see 8.2
4. Statement see 9.1
5. Block see 9.1

Semantics

Loop Variables

A `for` statement defines one or more *loop variables* that are given successive values from sequences during the execution of the loop. A loop variable definition in a `for` statement is thus the assigned source for the loop variable name so defined within the `for` statement. The name of a loop variable must be unassigned before the `for` statement, may not be otherwise assigned within the `for` statement, and is considered unassigned after the `for` statement.

There are two forms for a loop variable definition.

In the first form, the type of the loop variable is given implicitly by the expression from whose result the loop variable takes values. For example, suppose that the name `memberList` has the type `Member` and a multiplicity of `[0..*]` in the statement

```
for (member in memberList) {
  names->add(member.name);
  addresses->add(member.address);
}
```

The loop variable member is then implicitly given the type `Member`, the same type as that of the name expression `memberList`.

However, *collection conversion* may be performed on the expression in a loop variable definition. That is, if the type of the expression is a collection class (see 11.7) and the multiplicity upper bound of the expression is no greater than 1, then the operation `toSequence` is implicitly called on the expression to produce the sequence of values for the loop variable. The loop variable is then the type of the result of the `toSequence` call (i.e., the element type of the collection). Thus, the example above would still be legal if the type of memberList was, e.g., `List<Member>`.

In the second form, the type of the loop variable is declared explicitly. For example, the following statement is semantically equivalent to the example given above:

```
for (Member member: memberList) {
  names->add(member.name);
  addresses->add(member.address);
}
```

In this form, the declared type of the loop variable normally must conform to the type of the expression (see 8.2 for the definition of type conformance). However, if collection conversion applies to the expression, then the declared type must conform to the element type of the collection.

NOTE. The second form of loop variable definition has a similar syntax to that used in a "for each" statement in Java.

A loop variable definition of the form `var in expr1..expr2` is a shorthand for `var in Integer[] {expr1..expr2}` (see 8.3.15 on ranges in sequence construction expressions). In this case, both expressions must have a type that conforms to type `Integer` and a multiplicity upper bound of 1. A loop variable so defined takes on sequential integer values in each loop iteration, beginning with the value given by *expr1* and ending with the value given by *expr2*. If the second value is less than the first, then the `for` loop execution completes with no iterations.

Iterative Execution

When a `for` statement is executed, the expressions in its loop variable definitions are evaluated sequentially. If the result for the first variable is a sequence of at least one value, and the `for` statement has no `while` condition expression, then the block of the `for` statement is executed once for each value in the sequence. On each such execution, the local names of the loop variables within the block have corresponding values from the sequences resulting from the evaluation of the expressions in their definitions.

Note that the sequences for all the loop variables in a specific execution of a `for` statement should have the same size. However, if this is not true, it is the sequence for the first variable that controls the execution. If another variable has a sequence of a larger size than the first variable, then the additional values will be ignored. If it has a sequence of a smaller size, then the variable will be empty on iterations after all values in its sequence have been used. For this reason, the first loop variable has the multiplicity [1..1], while any other loop variables are given the multiplicity `[0..1]`.

NOTE. At the minimum conformance level (see 2.2), a `for` statement is allowed to have only one loop variable.

By default, the executions of the body block of a `for` statement occur in sequential iterations. Values are taken from the sequence for each loop variable in order.

Since a loop variable expression is only evaluated once, any local names defined in it have the values assigned during that evaluation for all executions of the body block of the `for` statement, unless reassigned within that block.

Local names defined within the body block of a `for` statement are prohibited from being accessed outside the `for` statement, since they could be uninitialized if the `for` statement does not execute its body. Local names defined within an expression in a loop variable definition are available after completion of execution of the `for` statement.

Parallel Execution

If a for statement is preceded by a `@parallel` annotation (see 9.2), then the executions of the body block of the `for` statement occur concurrently rather than sequentially.

A `@parallel` annotation may include a list of names. Each such name must be already assigned before the body of the for statement, with a multiplicity upper bound other than 1. They may then be used within the body block of the `for` statement to collect values across the concurrent executions of that block (any prior assigned values are lost). As such, these names may only appear within the `for` statement as the target argument (i.e., argument to the first parameter) for the `add` function from the Alf `CollectionsFunctions` library package (see 11.6).

For example, consider the following piece of a Fast Fourier Transform (FFT) computation, using an iterative `for` statement.

```
Sn_Even = Integer[]{};
Sn_Odd = Integer[]{};

for (lower in S_Lower, upper in S_Upper, root in V) {
  Sn_Even -> add(lower+upper);
  Sn_Odd  -> add((lower-upper)*root);
}
```

This is computationally correct, but it forces sequential execution of what is essentially a parallel algorithm. This can be corrected by inserting appropriate `@parallel` annotations:

```
Sn_Even = Integer[]{};
Sn_Odd = Integer[]{};

//@parallel(Sn_Even,Sn_Odd)
for (lower in S_Lower, upper in S_Upper, root in V) {
  //@parallel
  {
    Sn_Even -> add(lower+upper);
    Sn_Odd  -> add((lower-upper)*root);
  }
}
```

The listing of the names `Sn_Even` and `Sn_Odd` in the `@parallel` annotation is required so that the `add` operation invocations continue to be permitted within the `for` statement.

NOTE. The normal behavior invocation notation (see 8.3.9) may also be used for the add function instead of the sequence operation notation (see 8.3.17). For example "`Sn_Even->add(lower+upper);`" may instead be written "`add(Sn_Even, lower+upper);`". The behavior invocation notation is available at the minimum conformance level, while the sequence operation notation is not available until the full conformance level (see also 2.2 on syntactic conformance levels).

If, after the loop variable definitions of a parallel `for` statement, a name has an assigned source, then it must have the same assigned source after the block of the for statement. Other than for names defined in the `@parallel` annotation of the `for` statement, the assigned source for such names is the same after the

`for` statement as before it. Any names defined in the `@parallel` annotation have the `for` statement itself as their assigned source after the `for` statement.

Unlike the case of an iterative `for` statement, names defined before a parallel `for` statement may *not* be reassigned within the statement, unless they are listed in the `@parallel` annotation for the statement. As in the iterative case, any names defined within the body of a parallel `for` statement are not available outside of the `for` statement.

9.13 `break` Statements

A `break` statement completes execution of an enclosing `switch`, `while`, `do` or `for` statement.

Syntax

```
BreakStatement(s: BreakStatement)
      = "break" ";"
```

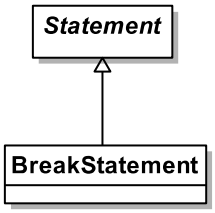

Figure 9.14 Abstract Syntax of `break` Statements

Cross References

1. Statement see 9.1

Semantics

A `break` statement may only be used directly or indirectly within the body of a `switch`, `while`, `do` or `for` statement (sees 9.9, 9.10, 9.11 and 9.12, respectively), except that the innermost enclosing such statement must not be a `for` statement with a `@parallel` annotation. When a `break` statement is executed, it transfers control to the innermost enclosing `switch`, `while`, `do` or `for` statement, which then immediately completes normally.

9.14 `return` Statements

If an activity has a `return` parameter, then a `return` statement may be used to provide a value for that parameter and terminate execution of the activity.

Examples

```
return item;
```

```
return list[index];
```

```
return x * factorial(x-1);
```

Syntax

```
ReturnStatement(s: ReturnStatement)
      = "return" [ Expression(s.expression) ] ";"
```

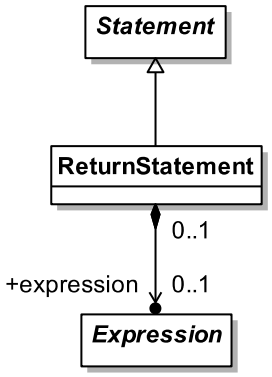

Figure 9.15 Abstract Syntax of `return` Statements

Cross References

1. Expression see 8.1
2. Statement see 9.1

Semantics

A return statement is used to terminate the execution of the enclosing behavior. If the enclosing behavior has a return parameter, then the return statement must have an expression that is assignable to the return parameter (see 8.8 for the definition of *assignability*). Otherwise, it must not have an expression.

When a `return` statement is executed, , if it has an expression, that expression is evaluated, and the resulting values (if any) are assigned to the return parameter. The execution of the enclosing behavior then immediately terminates.

9.15 `accept` Statements

An `accept` statement is used to accept the receipt of one or more types of signals.

NOTE. `accept` statements are not available at the minimum conformance level (see 2.2).

Examples

```
accept (sig: SignalNewArrival, SignalTermination);
```

```
accept (arrival: SignalNewArrival) {
  WriteLine(arrival.name);
  terminate = false;
} or accept (SignalTermination) {
  terminate = true;
}
```

Syntax

```
AcceptStatement(s: AcceptStatement)
      = SimpleAcceptStatement(s)
      | CompoundAcceptStatement(s)
SimpleAcceptStatement(s: AcceptStatement)
      = AcceptClause(s.acceptBlock) ";"
CompoundAcceptStatement(s: AcceptStatement)
      = AcceptBlock(s.acceptBlock) { "or" AcceptBlock(s.acceptBlock) }
AcceptBlock(a: AcceptBlock)
      = AcceptClause(a) Block(a.block)
```

```
AcceptClause(a: AcceptBlock)
      = "accept" "(" [ Name(a.name) ":" ] QualifiedNameList(a.signalNames)
        ")"
QualifiedNameList(qList: QualifiedNameList)
      = QualifiedName(qList.name) { "," QualifiedName(qList.name) }
```

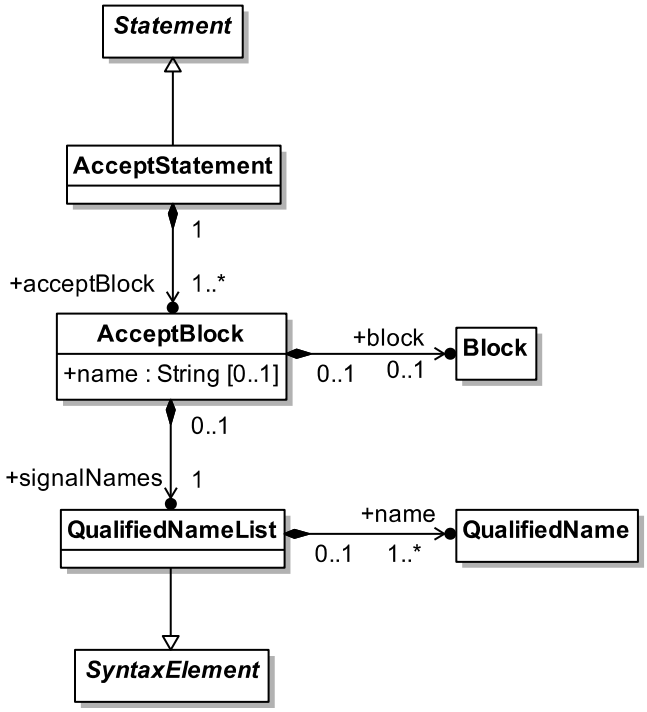

Figure 9.16 Abstract Syntax of `accept` Statements

Cross References

1.	Name	see 7.6
2.	Expression	see 8.1
3.	QualifiedNameList	see 8.2
4.	Statement	see 9.1
5.	Block	see 9.1

Semantics

An `accept` statement can only be used within the definition of an active behavior or the classifier behavior of an active class. All listed qualified names in an accept clause of an `accept` statement must resolve to signals for which the enclosing behavior has a reception. These signals must not be templates, though they can be the bindings of template signals (see 8.2). No signal may be named in more than one accept clause of an `accept` statement.

Simple `accept` Statements

In its simplest form, an accept statement simply identifies a signal by name:

```
accept (SignalNewArrival);
```

When this statement is executed, the thread of execution it is on is suspended, waiting for the receipt of an instance of the signal `SignalNewArrival`. When such a receipt later triggers the `accept` statement, it completes its execution, and further execution on its thread can continue.

An `accept` statement can also optionally include the definition of a local name that is used to hold an accepted signal instance:

```
accept (arrival: SignalNewArrival);
WriteLine(arrival.name);
```

A local name so defined has the signal as its type. It must be unassigned before the `accept` statement.

A single `accept` statement can list multiple signals, any one of which may trigger it. If the accept statement includes a local name, then this local name will hold whichever signal instance is actually received. The type of such a local name is the effective common ancestor of the listed signals (as defined in 8.7), if one exists, and is untyped otherwise.

For example:

```
accept (sig: SignalNewArrival, SignalTermination);
if (sig instanceof SignalNewArrival) {
  WriteLine(((SignalNewArrival)sig).name);
  terminate = false;
} else {
  terminate = true;
}
```

Compound `accept` Statements

In the example above, which of multiple signals was actually received is determined by testing the type of the instance received. This is such a typical pattern that Alf provides a *compound* version of the `accept` statement that allows the explicit specification of separate clauses for the receipt of different signals. Each clause is triggered on a specific set of signals and, if any one of them is received, an associated block is executed, after which execution of the `accept` statement is complete.

For example, the above example can be rewritten as follows using a compound `accept` statement:

```
accept (arrival: SignalNewArrival) {
  WriteLine(arrival.name);
  terminate = false;
} or accept (SignalTermination) {
  terminate = true;
}
```

For a compound `accept` statement, a local named defined in an accept clause (such as `arrival` in the example above) is available only in the body of that clause and is considered unassigned after the `accept` statement. New local names may be defined within the `accept` statement (such as `terminate` in the example above). After the `accept` statement, such a new local name has a type that is the effective common ancestor (see the definition in 8.7) of the type of the name in each accept clause in which it is defined with a multiplicity lower bound that is the minimum of the lower bound for the name in each clause and a multiplicity upper bound that is the maximum of the upper bound for the name in each clause. If the name is not defined in every clause of the `accept` statement, then it is considered to have multiplicity lower and upper bounds of 0 for the purposes of the above bounds determination. If the clause of the `accept` statement that is executed does not define a name that is defined in a different clause, then that name is defined but empty after the execution of the `accept` statement.

9.16 `classify` Statements

A `classify` statement is used to dynamically reclassify an already existing object. The statement identifies an already existing object and the classes from which and/or to which the identified object is to be reclassified.

NOTE. `classify` statements are not available at the minimum conformance level (see 2.2).

Examples

```
classify principal from * to Administrator;
```

```
classify principal from Administrator;
```

```
classify monitor from InActiveMonitor to ActiveMonitor;
```

```
classify this
  from Pending, Overdue
  to Resolved, InProcess;
```

Syntax

```
ClassifyStatement(s: ClassifyStatement)
      = "classify" Expression(s.expression) ClassificationClause(s) ";"
ClassificationClause(s: ClassifyStatement)
      = ClassificationFromClause(s.fromList)
        [ ClassificationToClause(s.toList) ]
      | [ ReclassifyAllClause (s.isReclassifyAll=true) ]
        ClassificationToClause(s.toList)
ClassificationFromClause(qList: QualifiedNameList)
      = "from" QualifiedNameList(qList)
ClassificationToClause(qList: QualifiedNameList)
      = "to" QualifiedNameList(qList)
ReclassifyAllClause(qList: QualifiedNameList)
      = "from" "*"
```

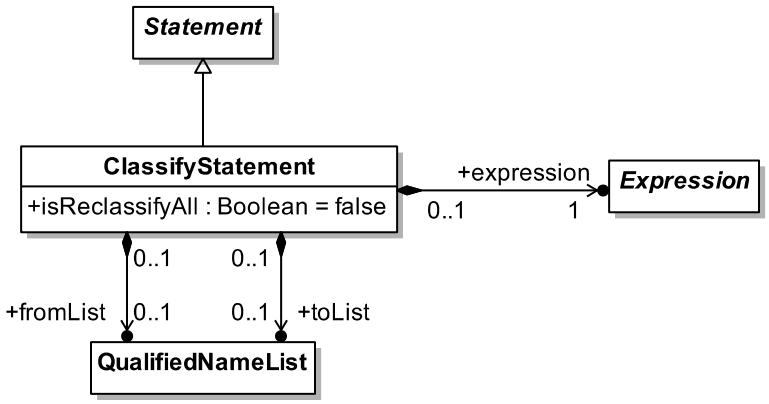

Figure 9.17 Abstract Syntax of `classify` Statements

Cross Reference

1. Expression see 8.1
2. Statement see 9.1
3. QualifiedNameList see 9.15

Semantics

The target expression in a `classify` statement must have a class as its static type. All qualified names listed in the `from` or `to` lists must resolve to classes. All the classes in the `from` and `to` lists must be subclasses of the static type of the target expression and none of them may have a common superclass that is a subclass of the static type of the target expression (that is, they must be *disjoint* subclasses).

NOTE. The restriction on reclassification to be only between disjoint subclasses allows type safety to be maintained if all potentially reclassifiable objects are only referenced via their superclass interface. However, it is still possible to downcast such an object to a subclass, and type safety may be violated if that subclass is later removed from the object via some other reference.

The target expression of a `classify` statement must evaluate to a single object. When the `classify` statement completes execution, the object is no longer classified by the classes in the `from` list (unless they are also in the `to` list) and is classified by all the classes in the `to` list.

If the `from` list is given as "`*`", then all the current classes of the identified object are removed and replaced with the classes in the `to` list. In this case, the `to` list must not be empty.

Neither destructor nor constructor operations are called during reclassification, and any initializers on attributes of classes in the `to` list are *not* evaluated. All new attributes are initialized as empty, even if this violates their declared multiplicity. However, if any of the classes on the `to` list are active classes with classifier behaviors, then these behaviors *are* started for the target object (except that any such behavior that is already running from the previous classification is not restarted but simply continues to run).

10 Units

10.1 Overview

Alf adds the concept of a *unit* to the basic UML concepts of namespaces and packages. A unit is a namespace defined using Alf notation that is not itself textually contained in any other Alf namespace definition.

This clause describes how structural models (largely within the fUML subset) can be represented textually as Alf units. This includes the notation for defining the kinds of *classifiers* included in the fUML subset (see 10.4) and for defining *packages* to group the definition of other elements (see 10.3). Subclause 10.5 discusses the representation of various kinds of features of classifiers.

The structural modeling constructs defined in this clause are only included at the *extended compliance* level of Alf (see 2.2 on the definition of syntactic conformance).

Units are lexically independent (though semantically related) segments of Alf text that provide a level of granularity similar to typical programming language text files. A unit may also have *subunits* that define namespaces that are owned (directly or indirectly) by the unit but whose Alf definition is given by a unit that is textually separate from the base unit. Inclusion in the base unit is indicated using a *stub declaration* in the base unit and a *namespace declaration* in the definition of the subunit.

Since an Alf unit can be processed into a UML abstract syntax representation, a portion of a model represented in Alf can always be integrated into a larger model on that basis, regardless of the surface representation of any portion of the model.

Examples

```
private import ProductSpecification::Product;    // Element import
private import EE_OnlineCustomer as OL_Customer; // Element import with alias
private import DomainDataTypes::*;               // Package import
package Ordering                                 // Unit definition
{
  class Order;                                   // Stub declaration
}

namespace Ordering;                              // Namespace declaration
public import Customer;                          // Public element import
/** Order class of the Ordering subsystem. */    // Documentation comment
class Order {                                    // Subunit definition
  ...
}

@apply(DataProfile)                              // Profile application
@Entity(tableName=>"CustomerTable")              // Stereotype annotation
class Customer {
  ...

}
```

Syntax

```
UnitDefinition(u: UnitDefinition)
      = [ NamespaceDeclaration(u.namespaceName) ]
        { ImportDeclaration(u.import) }
        [ DocumentationComment(u.documentation) ]
        { StereotypeAnnotation(u.definition.annotation) }
        NamespaceDefinition(u.definition)
NamespaceDeclaration(q: QualifiedName)
      = "namespace" QualifiedName(q) ";"
ImportDeclaration(i: ImportReference)
      = ImportVisibilityIndicator(i.visibility) "import"
        ImportReference(i) ";"
ImportVisibilityIndicator(v: String)
      = "public"(v) | "private"(v)
ImportReference(i: ImportReference)
      = ElementImportReference(i)
      | PackageImportReference(i)
ElementImportReference(i: ElementImportReference)
      = QualifiedName(i.referentName) [ AliasDefinition(i.alias) ]
AliasDefinition(n: String)
      = "as" Name(n)
PackageImportReference(i: PackageImportReference)
       = ColonQualifiedName(i.referentName) "::" "*"
      | DotQualifiedName(i.referentName) "." "*"
      | UnqualifiedName(i.referentName) "::" "*"
      | UnqualifiedName(i.referentName) "." "*"
StereotypeAnnotation(s: StereotypeAnnotation)
      = "@" QualifiedName(s.stereotypeName) [ "(" TaggedValues(s) ")" ]
TaggedValues(s: StereotypeAnnotation)
      = QualifiedNameList(s.names)
      | TaggedValueList(s.taggedValues)
TaggedValueList(t: TaggedValueList)
      = TaggedValue(t.taggedValue) { "," TaggedValue(t.taggedValue) }
TaggedValue(t: TaggedValue)
      = Name(t.name) "=>" LiteralValue(t)
LiteralValue(t: TaggedValue)
      = BooleanLiteral(t.value)
      | [ NumericUnaryOperator(t.operator) ] NaturalLiteral(t.value)
      | [ NumericUnaryOperator(t.operator) ] RealLiteral(t.value)
      | UnboundedValueLiteral(t.value)
      | StringLiteral(t.value)
```

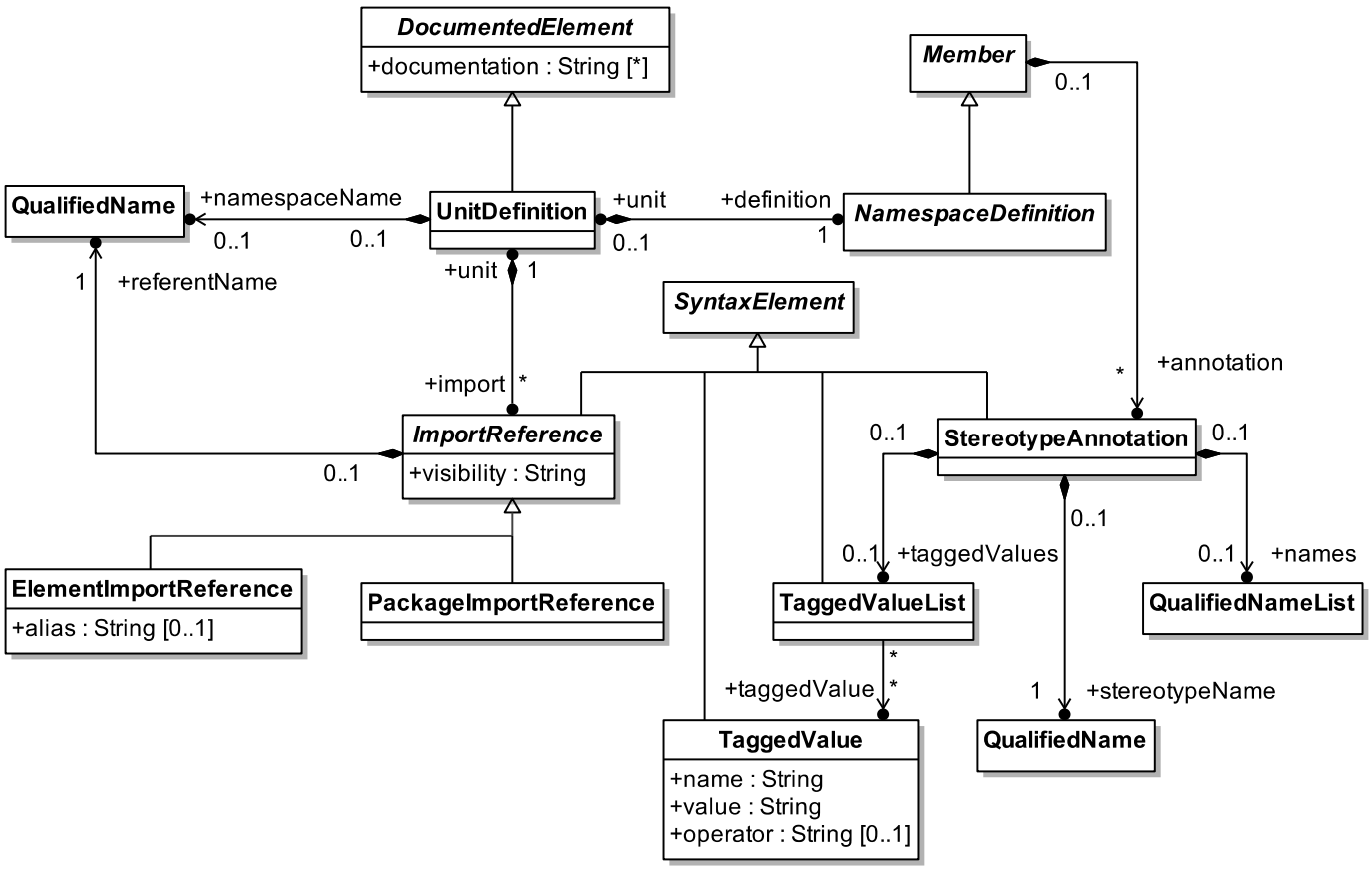

Figure 10.1 Abstract Syntax of Unit Definitions

Cross References

1. DocumentedElement see 6.6
2. DocumentationComment see 7.5.3
3. Name see 7.6
4. BooleanLiteral see 7.8.2
5. NaturalLiteral see 7.8.3
6. UnboundedValueLiteral see 7.8.4
7. StringLiteral see 7.8.5
8. RealLiteral see 7.8.6
9. QualifiedName see 8.2
10. ColonQualifiedName see 8.2
11. DotQualifiedName see 8.2
12. UnqualifiedName see 8.2
13. NumericUnaryOperator see 8.5.4
14. QualifiedNameList see 9.15

15. NamespaceDefinition see 10.2

16. Member see 10.2

Semantics

Subunits

Each kind of namespace definition (as given in subsequent subclauses) includes the specification of a name for the namespace, which, for a unit definition, becomes the name of the unit. If the unit definition has a namespace declaration, then the unit is an owned member of that namespace. If the declared namespace is contained itself in an Alf unit, then the unit definition is for a subunit and the declared namespace must have a stub declaration for the subunit. A namespace is always denoted in a namespace declaration by its fully qualified name (as defined in 8.2).

For example, the following definition for the package `Ordering` has a stub declaration for the class `Order`.

```
package Ordering {
  class Order;       // Stub declaration
}
```

The subunit defining class `Order` must then have a corresponding namespace declaration indicating that it is completing a stub in the namespace `Ordering`.

```
namespace Ordering;  // Namespace declaration
class Order {        // Subunit definition
  …
}
```

(See the subclause for each kind of namespace definition for further discussion of stub declarations.)

Model Units

An Alf *model unit* is an Alf unit that is not a subunit of any other Alf unit. It may be used to represent the model of a classifier or package that is intended to be individually referenced as a named element.

A model unit is not required to have a namespace declaration. But, if it does have such a declaration, then, by definition, the referenced namespace will not be represented using Alf. If it does not have a namespace declaration, then which namespace it is placed in, if any, is tool specific.

A model unit may represent an entire UML model (at least within the limits of the fUML subset) or it may represent a model element (such as a class or standalone activity) intended to be used within some larger model. The Alf specification does not define how such an Alf unit is created within a specific modeling environment or how it is attached to some larger model within the environment. It does, however, place requirements on the modeling environment to allow references from within Alf units to named elements defined in namespaces outside of those units (see 10.2).

Import Declarations

An *import declaration* specifies a UML import dependency (see UML Superstructure, subclauses 7.3.15 and 7.3.39). Such a declaration may specify an *element import* or a *package import,* and it may be *public* or *private.* Alf notation only provides for import declarations on namespaces defined as units. The import declarations for a unit are placed after the namespace declaration (if any) and before any stereotype annotations and the body of the definition for the unit. The qualified name given in an import declaration must be a fully qualified name (as defined in 8.2).

For an element import reference:

- The name must resolve to a packageable element with either an empty or public visibility.
- The visibility of the named element within the scope of the unit definition is as specified in the UML Superstructure, 7.3.15, ElementImport.

For a package import reference:

- The name must resolve to a package with either an empty or public visibility.
- The visibility of the named element within the scope of the unit definition is as specified in the UML Superstructure, 7.3.39, PackageImport.

For example, the following is a private element import declaration.

```
private import ProductSpecification::Product;    // Element import
package Ordering {                               // Namespace definition
  …
}
```

This declaration specifies that the element with the qualified name `ProductSpecification::Product` be included as a member of the namespace `Ordering`. Since this is a private import, the element is imported as a private member and is not visible outside `Ordering`. On the other hand, a public import such as

```
public import ProductSpecification::Product;     // Public element import
package Ordering {                               // Namespace definition
  …
}
```

specifies that the element be imported as a public member of `Ordering`. In this case, the qualified name `Ordering::Product` refers to the same element as `ProductSpecification::Product`.

An element may also be imported with an *alias*. For example:

```
public import ProductSpecification::Product as Prod; // Import with alias
package Ordering {                                   // Namespace definition
  …
}
```

In this case, within the `Ordering` namespace, the unqualified name for `ProductSpecification::Product` is `Prod`, *not* `Product`. Further, since the import is public, outside `Ordering` the qualified name `Ordering::Prod` can be used to refer to the same element as `ProductSpecification::Product`.

A package import declaration may be used to import all the elements of a package. For example, the following declaration:

```
private import ProductSpecification::*;          // Package import
package Ordering {                               // Namespace definition
  …
}
```

specifies that all the elements of the package `ProductSpecification` should be imported into the namespace `Ordering` as private members. This is equivalent to giving a private element import declaration for each of the elements in `ProductSpecification`. As with an element import declaration, a package import may also be public. However, aliases cannot be defined with a package import.

NOTE. The UML semantics of importing an element is simply that it becomes a member of the importing namespace. A namespace is not required to have an import dependency in order to reference

an external element from within the namespace. In Alf, importing an element means the name of the imported element is in the current scope within the importing namespace and can, therefore, be used without qualification (see 8.2). However, it is always possible to refer to an element using its fully qualified name (if such exists) *without* having to import the element.

Unless it is stereotyped as a «ModelLibrary», an Alf model unit has an implicit, private package import for each of the sub-packages of the `Alf::Library` package (see 11.1). These packages must therefore be available in the modeling environment for any Alf unit. (Note that it is unnecessary for subunits to have such implicit imports, since they will have visibility to the imports on the enclosing model unit.)

The standard «ModelLibrary» stereotype can be applied to a package to indicate that it "contains model elements that are intended to be reused by other packages" (see UML Superstructure, Annex C). This stereotype can be applied to a package represented in Alf using a `@ModelLibrary` annotation (see below). A model-unit package so annotated does *not* have the implicit imports for the `StandardModelLibrary` packages and must import these explicitly if they are needed.

NOTE. This rule is to allow for the minimization of dependencies of model library packages represented in Alf on other packages that they may not need. It also allows for the possibility of representing the `StandardModelLibrary` as a model library package in Alf without requiring it to implicitly, circularly import subpackages of itself.

Stereotype Annotations

A *stereotype annotation* specifies the application of a stereotype to a unit. Such annotations are listed immediately before the namespace definition for the unit. In general, the qualified name in a stereotype application must resolve to a stereotype in a profile applied to some (directly or indirectly) enclosing package. The unit must be of a syntactic type that corresponds to the metaclass extended by the identified stereotype. Any stereotype may be applied at most once to a unit.

NOTE. Profiles and stereotypes are not included in the fUML subset. However, allowing stereotype application in Alf provides an extensible annotation mechanism based on profiles. Modeling tools may use such annotations to implement profile-specific mappings of Alf text to the strict fUML subset for execution. Note, however, that Alf does *not* provide a textual notation for the definition of profiles, only their application.

The stereotype name does not need to be qualified if there is only one applied profile with a stereotype of the given name (except for the special cases given in Table 10.1 and discussed further below). Otherwise the stereotype must be fully qualified with the name of the profile. However, the UML superstructure standard profiles (see UML Superstructure, Annex C) are considered to be implicitly applied, and stereotypes from these profiles can always be used without explicit qualification of their names. Thus, any stereotype with a name that conflicts with the name of a standard stereotype must always be fully qualified.

A stereotype annotation may optionally be followed by a list of *tagged values.* There are two forms for such a list.

The first form has a syntax similar to that of a named tuple (see 8.3.8), with a value associated with each attribute of the stereotype. Optional attributes (multiplicity lower bound of 0) may be omitted. This Alf syntax only allows for tagged values of a primitive type and only one value per attribute. Therefore, this form of annotation can only be used for stereotypes with attributes that have one of the standard primitive types (`Boolean`, `String`, `Integer`, `Real` or `UnlimitedNatural`) and a multiplicity lower bound of at most 1.

The second form of tagged value list is a list of qualified names. In this case, the stereotype must have a single attribute whose type is a metaclass from the UML abstract syntax. The listed names must be fully qualified and resolve in model scope for the unit to model elements consistent with the required metaclass for the stereotype (see 10.2 for the definition of model scope).

Alf also uses the annotation syntax for a limited number of cases other than strict stereotype application, as shown in Table 10.1. In the case that there are visible stereotypes with the same name as these special case annotations, then the unqualified names always denote the special cases, not the stereotypes. To apply the stereotypes, they must be qualified with the name of the profile in which they are defined, whether or not that would otherwise be necessary.

Table 10.1 Non-Stereotype Annotations

Annotation	Applies To	Description
`@apply(`p_1`,…,`p_n`)`	Package Definition	Apply the profiles denoted by the qualified names p_1, …, p_n. The names must resolve to profiles.
`@primitive`	Data Type Definition or Activity Definition	If applied to a data type, indicates that the data type is to be registered as a primitive data type. The data type may not have any attributes. If applied to an activity, indicates that the activity is instead to be mapped as a primitive behavior. How the implementation of this primitive behavior is actually defined is implementation specific. Definitions marked as primitive may not be templates or stubs.
`@external (file=>"...")`	NamespaceDefinition	Must be applied to a stub declaration. Indicates that the stub is to be completed by a subunit implementation external to the normal UML/Alf modeling environment. How this implementation is attached to the UML model and what specific kinds of namespaces can be annotated as external are tool-specific. However, the `file` tag can *optionally* be used to give a file reference for the subunit implementation.

The first special case shown in Table 10.1 is for denoting the application of a profile. For cases other than the standard UML profiles, such a profile application annotation for the profile must appear before the stereotypes for the profile are used in stereotype annotations. Once a profile is applied to a unit, however, it is also available without further application to all subunits of that unit.

10.2 Namespaces

A *namespace* is a UML element used to provide the definition context for a set of named elements known as the *owned members* of the namespace. A *package* is a UML construct whose primary function is simply as a namespace for defining other elements (see 10.3). However, a *classifier* (class, structured

data type, enumeration, association, signal or activity) also acts as a namespace for various sub-elements defined within it (see 10.4). Generally, each kind of namespace has specific restrictions on the kinds of named elements that can be owned members, as reflected in the Alf notation for them described in subsequent subclauses.

In UML, an operation is also considered to be a namespace for its parameters, and it can be used as such in the qualified name for those parameters (see 8.2). However, an operation can only be defined textually within the context of a class (sees 10.4.2 and 10.5.3). Therefore, the definition of an operation in Alf is *not* considered to syntactically be a namespace definition, in the same sense that package and classifier definitions are. In particular, an operation definition *cannot* be used as a unit (though an activity acting as the method for an operation *can* be used as a subunit completing an operation stub declaration—see 10.4.8).

Since a namespace is a named element, it may itself be an owned member of an *enclosing* namespace. In addition to its owned members, the *members* of a namespace include the members of any enclosing namespace (unless *hidden* by an owned member with the same name). A namespace may also have members that are *imported* from other packages.

Syntax

```
NamespaceDefinition(d: NamespaceDefinition)
      = PackageDefinition(d)
      | ClassifierDefinition(d)
VisibilityIndicator(v: String)
      = ImportVisibilityIndicator(v)
      | "protected"(v)
```

NOTE. The actual definition of specific kinds of named elements allowed in various kinds of namespaces is given in the following subclauses. However, the syntax and semantics of visibility are discussed here, because they are largely common across the different kinds of named element definitions.

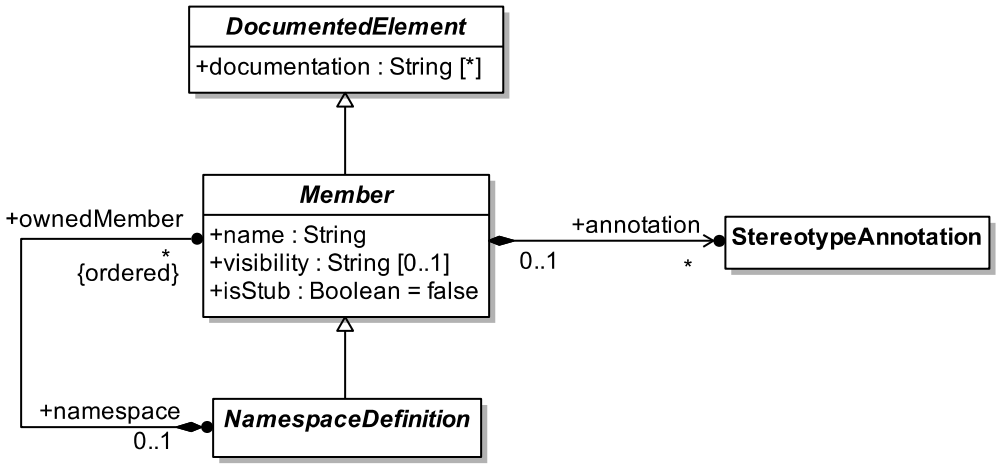

Figure 10.2 Abstract Syntax of Namespace Definitions

Cross References

1. DocumentedElement see 6.6
2. ImportVisibilityIndicator see 10.1

3. StereotypeAnnotation see 10.1
4. PackageDefinition see 10.3
5. ClassifierDefinition see 10.4

Semantics

Members

Each kind of namespace definition contains constituent definitions for owned members of the namespace. The members of a namespace must be distinguishable as specified in the UML Superstructure, 7.3.34, Namespace. However, in any case that the UML Superstructure considers two names to be distinguishable if they are different, an Alf implementation may instead impose the stronger requirement that the names not be conflicting, in the sense defined in 7.6 (of the Alf specification).

Model Scope

All owned members of an Alf namespace must be represented in Alf. However, Alf text may occur in the context of a larger UML model, not all of which is represented in Alf. In such a case, it is possible from within the Alf text to refer by name to named elements defined in the wider model context. For any Alf namespace, the *model scope* is the innermost namespace enclosing the Alf namespace that is not itself represented in Alf, if any. If there is no such namespace, then the Alf unit has an *empty* model scope.

Whether a name is *visible in the model scope* is expected to be determined using the usual UML Superstructure rules. At the very least, the names of all members of the model scope namespace should be visible. However, the management of namespaces at the model scope and any enclosing namespaces above that is the responsibility of the modeling environment and not otherwise defined in the Alf specification.

UML does not in general require that a named element be a member of a namespace or that there be only one hierarchy of namespaces. However, in order to be referenced in an Alf text, any model element external to that text must be nameable by a qualified name beginning with a name visible in the model scope.

Visibility

The visibility of a name outside the scope of the namespace owning the named element can be controlled by placing a *visibility indicator* on the definition of the named element: one of "`public`", "`private`" or "`protected`". A named element definition with no visibility indicator is considered to have package visibility.

The visibility of a named element outside its defining scope is as defined in the UML Superstructure, 7.3.55, VisibilityKind.

10.3 Packages

A *package* is a namespace whose sole function is to group its member elements, which must be *packageable elements*. In Alf, the supported kinds of packageable element definitions are just the namespace definitions for packages and the various kinds of classifiers (see 10.4). Note also that only packageable elements may be imported into other namespaces (see 10.1).

A packageable element may be fully defined within the textual body of a package definition. Alternatively, a *stub declaration* may be given for the element, which includes only the element name and visibility (and, for an activity, its signature). The full definition of the element is then given in a *subunit* definition (see 10.1).

Examples

```
package Ordering                                  // Base unit
{
  public assoc Selects                            // Nested namespace
  {
    public cart:             ShoppingCart[0..*];
    public selectedProducts: Product[1..*];
    public selectionInfo:    ProductSelection;
  }

  public active class ShoppingCart;               // Stub declaration
  public abstract active class Order;
  public class ProductSelection;
}
```

Syntax

```
PackageDeclaration(d: PackageDefinition)
      = "package" Name(d.name)
PackageDefinition(d: PackageDefinition)
      = PackageDeclaration(d) "{" { PackagedElement(d.ownedMember) } "}"
PackagedElement(m: Member)
      = [ DocumentationComment(m.documentation) ]
        { StereotypeAnnotation(m.annotation) }
        ImportVisibilityIndicator(m.visibility) PackagedElementDefinition(m)
PackagedElementDefinition(m: Member)
      = NamespaceDefinition(m)
      | NamespaceStubDeclaration(m)
NamespaceStubDeclaration(m: Member)
      = PackageStubDeclaration(m)
      | ClassifierStubDeclaration(m)
PackageStubDeclaration(m: Member)
      = PackageDeclaration(m) ";" (m.isStub=true)
```

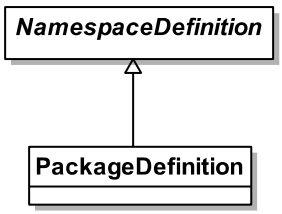

Figure 10.3 Abstract Syntax of Package Definitions

Cross References

1. DocumentationComment see 7.5.3
2. Name see 7.6
3. StereotypeAnnotation see 10.1
4. NamespaceDefinition see 10.2
5. ImportVisibilityIndicator see 10.2
6. ClassifierStubDeclaration see 10.4

Semantics

The package being defined is the current scope for all packaged element definitions within it.

Stereotype annotations apply to the element defined by the following packaged element definition. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing package, rather than in model scope. If the package element definition is a stub declaration, then the annotation for a stereotype may be applied either to the stub declaration or the subunit definition, but not both.

See also the discussion of the semantics for namespaces in general in 10.2.

10.4 Classifiers

10.4.1 Overview

A *classifier* specifies a classification of instances according to their features. Classifiers may participate in generalization relationships, which can result in its feature elements being inherited. Alf supports the following kinds of classifier: classes (including active classes), structured data types, enumerations, associations, signals and activities. This subclause specifies how each kind of classifier may be defined in Alf.

Syntax

```
ClassifierDefinition(d: ClassifierDefinition)
      = ClassDefinition(d)
      | ActiveClassDefinition(d)
      | DataTypeDefinition(d)
      | EnumerationDefinition(d)
      | AssociationDefinition(d)
      | SignalDefinition(d)
      | ActivityDefinition(d)
ClassifierDeclaration(d: ClassifierDefinition)
      = ClassDeclaration(d)
      | ActiveClassDeclaration(d)
      | DataTypeDeclaration(d)
      | EnumerationDeclaration(d)
      | AssociationDeclaration(d)
      | SignalDeclaration(d)
      | ActivityDeclaration(d)
ClassifierStubDeclaration(d: ClassifierDefinition)
      = ClassifierDeclaration(d) ";" (d.isStub=true)
ClassifierSignature(d: ClassifierDefinition)
      = Name(d.name) [ TemplateParameters(d) ]
        [ SpecializationClause(d.specialization) ]
TemplateParameters(d: ClassifierDefinition)
      = "<" ClassifierTemplateParameter(d.ownedMember)
        { "," ClassifierTemplateParameter(d.ownedMember) } ">"
ClassifierTemplateParameter(p: ClassifierTemplateParameter)
      = [ DocumentationComment(p.documentation) ] Name(p.name)
        [ TemplateParameterConstraint(p.specialization) ]
        (p.visibility="private" and p.isAbstract=true)
TemplateParameterConstraint(qList: QualifiedNameList)
      = "specializes" QualifiedName(qList.name)
```

```
SpecializationClause(qList: QualifiedNameList)
      = "specializes" QualifiedNameList(qList)
```

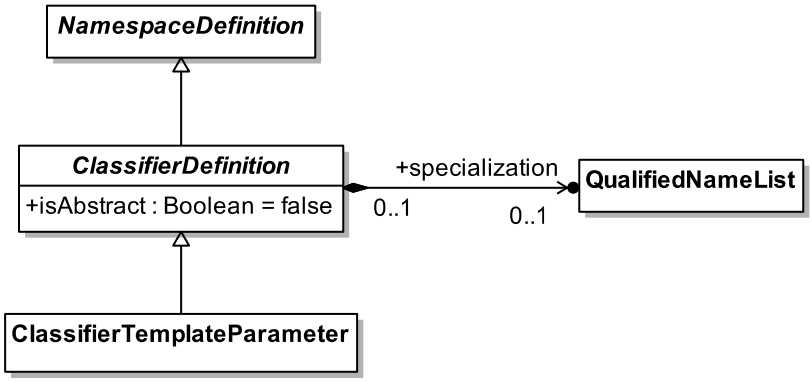

Figure 10.4 Abstract Syntax of Classifier Definitions

Cross References

1. DocumentationComment see 7.5.3
2. Name see 7.6
3. QualifiedNameList see 9.15
4. NamespaceDefinition see 10.2
5. ClassDefinition see 10.4.2
6. ClassDeclaration see 10.4.2
7. ActiveClassDefinition see 10.4.3
8. ActiveClassDeclaration see 10.4.3
9. DataTypeDefinition see 10.4.4
10. DataTypeDeclaration see 10.4.4
11. AssociationDefinition see 10.4.5
12. AssociationDeclaration see 10.4.5
13. EnumerationDefinition see 10.4.6
14. EnumerationDeclaration see 10.4.6
15. SignalDefinition see 10.4.7
16. SignalDeclaration see 10.4.7
17. ActivityDefinition see 10.4.8
18. ActivityDeclaration see 10.4.8

Semantics

As a namespace, a classifier includes definitions of features as elements. The detailed semantics for each kind of classifier are described in following sections of this subclause.

The semantics of classifiers are primarily static. However, a classifier provides the specification for creating instances that have execution semantics, as noted in the following sections of this subclause.

Specialization

In general, a classifier may *specialize* other classifiers of the same kind. That is, classes may specialize classes, data types may specialize data types, etc. If a classifier specializes one or more other classifiers, then the names of the classes it specializes are listed in a *specialization clause* following the keyword "`specializes`".

If a classifier listed in a specialization clause is a template (see below), then it must have a template binding giving arguments for all template parameters (see 8.2). Alf does not provide a notation for specializing an uninstantiated template classifier.

A classifier *inherits* non-private members from the classifiers it specializes. The visibility of inherited members from the classifiers named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier. Further, when used as a type, a classifier *conforms* to any classifier that it specializes (see 8.2 for the full definition of type conformance).

Any additional rules related to specialization of specific kinds of classifiers are discussed in the following subclauses on each kind of classifier.

Template Parameters

Alf provides general notation for the binding of the parameters of the various kinds of templates allowed by UML (see 8.2). Also also provides a notation for defining templates, but only for template classifiers whose template parameters are for classifiers. The primary motivation for including this capability in Alf is to allow for the definition of parameterized types, such as those defined in the Alf standard library package `CollectionClasses` (see 11.7), and parameterized behaviors (which are kinds of classes in UML), such as those defined in the Alf standard library package `CollectionFunctions` (see 11.6).

A classifier template parameter may optionally be *constrained,* such that any valid argument for the parameter must conform to a given classifier. If a parameter is so constrained, then the constraining classifier is named in a specialization clause for that parameter.

For example, a template class with the signature `Sorter<T>`, then `T` could be bound to any classifier for `T`. However, if the signature was `Sorter<T specializes Comparable>`, then `T` could only be legally bound to a class that was a subclass (directly or indirectly) of the class `Comparable`.

NOTE. Classifier templates are specified in the UML Superstructure, 17.5.7, with classifier template parameters described in 17.5.8. For classifier template parameters mapped from Alf, the allowSubstitutable property is always false.

Within the body of the definition of a classifier with template parameters, the parameters may be used as types. If a parameter is not constrained, then it is treated as if it was a data type with no attributes or operations. If the parameter is constrained, then it is considered to be the same kind of classifier as its constraining classifiers and to specialize the constraining classifier. Template parameters are always considered to be abstract classifiers that may not be directly instantiated, since any parameter could be substituted with an actual argument that is abstract. Template parameters are not visible outside of the classifier definition.

NOTE. The allowance for substituting a classifier template parameter without a constraining classifier with a classifier of any kind, regardless of the kind of parameterable element the template has, is a semantic variation point given in UML Superstructure, 17.5.7. This allowance is necessary in order to

provide for parameterized types that may be instantiated with any kind of argument type. The notational default for an unconstrained classifier template parameter is specified in UML Superstructure, 17.5.8, to be that it is considered to have a parameterable element that is a class. However, presuming, instead, that it is an abstract data type, as given above, prevents it from being used in places that require a class, which would result in an ill-formed model if the parameter was substituted with a type other than a class. There are no instances in the Alf notation in which an abstract data type (with no attributes or operations) may be used, but some other kind of classifier could not be used in the same place.

When the template classifier is instantiated and its parameters are bound, the result is effectively an equivalent bound element in which all of the template parameters have been replaced with the arguments to which they are bound (see 6.3 on the copy semantics of templates). A template parameter without a constraining classifier may actually be bound to an argument that is any kind of classifier, not just a class, as long as this would not make the equivalent bound element ill formed. A template parameter with a constraining classifier must be bound to an argument classifier that conforms to the constraining classifier.

10.4.2 Classes

A *class* is a classifier whose instances are *objects.* The features of a class may include properties (see 10.5.2) and operations (see 10.5.3). (An active class may also have receptions as features—see 10.4.3.)

Examples

```
abstract class Selection {                    // Abstract class
  public abstract getSelectionValue(): Money;
                                              // Abstract operation definition
}

class ProductSelection                        // Concrete subclass
  specializes Selection {

  private quantity: Count;                    // Attribute definition
  private unitPriceOfSelection: Money;

  public select                               // Operation stub declaration
    (in cart: Cart, in product: Product, in quantity: Count);

  public getQuantity(): Count {               // Concrete operation definition
    return self.quantity;
  }

  public getUnitPriceOfSelection(): Money {
    return self.unitPriceOfSelection;
  }

  /** The total value is given by the         // Documentation comment
      quantity times the unit price. */
  public getSelectionValue(): Money {         // Redefined operation
    return self.getQuantity * self.getUnitPriceOfSelection;
  }

}

abstract class Collection<T>{ … }
```

```
class MapToString<T> specializes Map<Entry=>T, Value=>String> { }

class Sorter<T specializes Comparable> {

  private list: List<T>;
  @Create public Sorter() { }
  @Create public Sorter(list: List<T>);

  public append(elements: List<T>);
  public sort();
  public getList(): List<T>;
  public clear();
}
```

Syntax

```
ClassDeclaration(d: ClassDefinition)
      = [ "abstract" (d.isAbstract=true) ] "class" ClassifierSignature(d)
ClassDefinition(d: ClassDefinition)
      = ClassDeclaration(d) "{" { ClassMember(d.ownedMember) } "}"
ClassMember(m: Member)
      = [ DocumentationComment(m.documentation) ]
        { StereotypeAnnotation(m.annotation) }
        [ VisibilityIndicator(m.visibility) ] ClassMemberDefinition(m)
ClassMemberDefinition(m: Member)
      = ClassifierDefinition(m)
      | ClassifierStubDeclaration(m)
      | FeatureDefinition(m)
      | FeatureStubDeclaration(m)
```

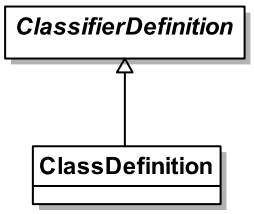

Figure 10.5 Abstract Syntax of Class Definitions

Cross References

1. DocumentationComment see 7.5.3
2. StereotypeAnnotation see 10.1
3. VisibilityIndicator see 10.2
4. ClassifierDefinition see 10.4.1
5. ClassifierStubDeclaration see 10.4.1
6. ClassifierSignature see 10.4.1
7. FeatureDefinition see 10.5.1
8. FeatureStubDeclaration see 10.5.1

Semantics

If the class definition is a subunit definition, then the definition of the namespace owning the class must include a class stub declaration for the class.

Class Members

The class being defined is the current scope for all class member definitions within it.

The properties of a class define attributes that may take on values of the appropriate type in objects of the class.

The operations of a class may be invoked on objects of the class using invocation expressions (see 8.3.10). The *method* for an operation (as representable in Alf) is an activity that provides the behavior for the operation. The method definition for an operation may either be included in the class definition, or the class definition may include a stub declaration for the operation, which is then completed in a subunit (see 10.1). Note that a subunit that completes an operation stub declaration must be an activity (see 10.4.8).

A class is a namespace (see 10.2), and all its features are namespace members. A class may also define *nested classifiers* as namespace members. For purposes of naming, such classifier definitions are identical to definitions made within a package (see 10.3). However, as members of the classifier namespace, they have visibility of private members of the classifier that would not be visible outside the classifier.

For example, activities are kinds of classifiers (see 10.4.8) and can, therefore, have definitions nested inside a class.

```
class Singleton {
  @Create private Singleton() {}

  public activity getSingleton(): Singleton {
    instance = Singleton.allInstances();
    if (instance -> isEmpty()) {
      instance = new Singleton();
    }
    return instance[1];
  }

}
```

In the above example, the activity `getSingleton` is nested in the class `Singleton`. Because of this, it has visibility to the private constructor for `Singleton`, which is not visible outside of the class.

Any member definition of a class may be preceded by a documentation comment (see 7.5.3) that is attached to the element being defined.

A member definition may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing class, rather than in model scope. If the class member definition is a stub declaration, then the annotation for a stereotype may be applied either to the stub declaration or the subunit definition, but not both.

Class Specialization

A class may *specialize* one or more other classes, in which case it *inherits* members from the classes it specializes (its *superclasses*). Each of the names in the specialization part of a class definition must

resolve to a class. The visibility of inherited members from the classes named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier.

All non-private members of superclasses are inheritable, except for operations redefined in the *subclass* (see 10.5.3). However, all members of a namespace must be *distinguishable.* It is therefore not legal to define a class to inherit members that are not distinguishable.

By default, two named elements are distinguishable if they are either different kinds of elements (e.g., a property as opposed to an operation) or they have different names. However, operations with the same name may be distinguished if they have different *signatures* (see 10.5.3). Such operations are said to be *overloaded.*

NOTE. Alf does not allow the redefinition of any kinds of class members other than operations, because this is the only kind of redefinition allowed in the fUML subset (see fUML Specification, 7.2.2). The constraint on distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. The default definition for distinguishability of named elements is given in 7.3.33 of the UML Superstructure. The rule for the distinguishability for operations is that defined for all behavioral features in 7.3.5 of the UML Superstructure. Note that there is no special rule for activities, so all activities in a given namespace must have different names, regardless of their signature (that is, activities cannot be overloaded).

A subclass inherits non-private nested classes from its superclasses as well as features. Thus, a subclass of the `Singleton` class given in the example above will inherit the public activity `getSingleton`, but not the private constructor.

NOTE. The general rules for inheritance of members are defined for all classifiers in 7.3.8 of the UML Superstructure. The exclusion of redefined members is given for classes in 7.3.7 of the UML Superstructure.

A class may be defined to be *abstract.* An abstract class cannot be instantiated, but it can be used as a superclass of other classes. Only abstract classes may have abstract operations (see 10.5.3), whether these operations are directly owned by the class or inherited. A class that is not abstract is known as a *concrete* class.

Class Instantiation

An object is created as an instance of a class using an instance creation expression (see 8.3.12). An object has referential value semantics (see fUML Specification, 8.3.2). That is, the equality of objects is based on their *identity,* not on the values of their properties (see also 8.6.6 on the semantics of equality).

10.4.3 Active Classes

An *active class* is one whose instances (*active objects*) have independent threads of control. The independent behavior of an active class is specified by its *classifier behavior*. When a contrast is necessary, a class that is non-active may be referred to as a *passive class.*

An active class may have attributes, operations and nested classifiers, just like a passive class (see 10.4.2). However, only an active class may have receptions (see 10.5.4) as features and only active objects may receive signals.

NOTE. An activity is a kind of class in UML, but activities are never explicitly declared as active in Alf. Instead, activities used as operation methods are always mapped as *not* active, while standalone activities are always mapped as active (sees 10.4.8 and 19.10).

Examples

```
active class Order {                      // An active class

  public dateOrderPlaced: Date;           // Attribute definitions
  public totalValue:      Money;
  public deliveryAddress: MailingAddress;
  public contactPhone:    TelephoneNumber;

  public receive CheckOut;                // Reception definitions
  public receive SubmitCharge;
  public receive PaymentDeclined;
  public receive PaymentApproved;
  public receive OrderDelivered;

} do Order_Behavior                       // Classifier behavior stub

abstract active class ProcessQueue {      // Abstract active class

  private busy: Boolean = false;

  public receive signal Wait {            // Signal reception definitions
    public process: Process;
  }
  public receive signal Release {}

  protected abstract enqueue(in process: Process); // Abstract operations
  protected abstract dequeue(): Process;
  protected abstract processesWaiting(): Boolean;

} do {                                    // In-line classifier behavior
  while (true) {
    accept (sig: Wait) {               // Accept statement for signals
      if (this.busy) {
        this.enqueue(sig.process);
      } else {
        sig.process.resume();
      }
    } or accept (Release) {
      if (this.processesWaiting()) {
        this.dequeue().resume();
      } else {
        this.busy = false;
      }
    }
  }
}

active class ProcessQueueImpl             // Concrete active subclass
  specializes ProcessQueue {

  private waitingProcesses: Process[*] ordered;

  private enqueue(in process: Process);   // Concrete operation redefinitions
  private dequeue(): Process;
  private processesWaiting(): Boolean;
```

```
}                                          // No additional behavior
```

Syntax

```
ActiveClassDeclaration(d: ActiveClassDefinition)
      = [ "abstract" (d.isAbstract=true) ] "active" "class"
        ClassifierSignature(d)
ActiveClassDefinition(d: ActiveClassDefinition)
      = ActiveClassDeclaration(d) "{" { ActiveClassMember(d.ownedMember) }
        "}" [ "do" BehaviorClause(d.classifierBehavior)
        (d.classifierBehavior.visibility="private")
        (d.ownedMember->includes(d.classifierBehavior)) ]
BehaviorClause(a: ActivityDefinition)
      = Block(a.body)
      | Name(a.name) (a.isStub=true)
ActiveClassMember(m: Member)
      = [ DocumentationComment(m.documentation) ]
        { StereotypeAnnotation(m.annotation) }
        [ VisibilityIndicator(m.visibility) ] ActiveClassMemberDefinition(m)
ActiveClassMemberDefinition(m: Member)
      = ClassMemberDefinition(m)
      | ActiveFeatureDefinition(m)
      | ActiveFeatureStubDeclaration(m)
```

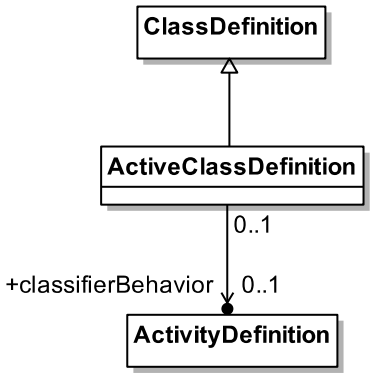

Figure 10.6 Abstract Syntax for Active Class Definitions

Cross References

1. DocumentationComment see 7.5.3
2. Name see 7.6
3. Block see 9.1
4. StereotypeAnnotation see 10.1
5. VisibilityIndicator see 10.2
6. ClassifierSignature see 10.4.1
7. ClassDefinition see 10.4.2
8. ClassMemberDefinition see 10.4.2
9. ActivityDefinition see 10.4.8
10. ActiveFeatureDefinition see 10.5.1

11. ActiveStubDeclaration see 10.5.1

Semantics

See also the discussion of the general semantics of classes in 10.4.2.

The classifier behavior of an active class is specified as an activity that is a private owned behavior of the class. This activity may be named in the definition of the active class, in which case the activity must be separately defined as a subunit of the active class with the given activity name. Alternatively, the activity may be specified with a block (see 9.1) directly attached to the active class definition. (Note that in neither case may the activity have parameters.)

An active class may specialize other classes, including passive classes, with the normal inheritance rules (see 10.4.2). However, a passive class may not specialize an active class.

Since the classifier behavior is always private, it is not inherited by subclasses. However, an instance of an active class with active superclasses will have the behavior specified for all its superclasses, as well as any behavior specified for the subclass. The classifier behavior for an active class may only accept signals for which the class has a reception, either directly or inherited from a superclass (see also 10.5.4 on receptions).

An active class may be *abstract* (see also 10.4.2), in which case it cannot be instantiated, but it can be used as the superclass of other active classes.

Active Class Instantiation

An active object is created like any other object using an instance creation expression. However, when an active object is created, its classifier behavior is automatically started (see 8.3.12).

10.4.4 Data Types

In Alf, the unqualified term *data type* is always used to refer to a *structured* data type, not an enumeration or primitive type. The instances of such a data type are known as *data values*. The features of a data type must be properties (see 10.5.2).

Examples

```
datatype Complex {
  public re: Real;
  public im: Real;
}
```

Syntax

```
DataTypeDeclaration(d: DataTypeDefinition)
      = [ "abstract" (d.isAbstract=true) ] "datatype" ClassifierSignature(d)
DataTypeDefinition(d: DataTypeDefinition)
      = DataTypeDeclaration(d) "{" { StructuredMember(d.ownedMember) } "}"
StructuredMember(m: Member)
      = [ DocumentationComment(m.documentations ]
        { StereotypeAnnotation(m.annotations) } [ "public"(m.visibility) ]
        PropertyDefinition(m)
```

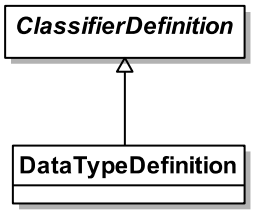

Figure 10.7 Abstract Syntax of Data Type Definitions

Cross References

1. DocumentationComment see 7.5.3
2. Name see 7.6
3. StereotypeAnnotation see 10.1
4. ClassifierDefinition see 10.4.1
5. ClassifierSignature see 10.4.1
6. PropertyDefinition see 10.5.2

Semantics

If the data type definition is a subunit definition, then the definition of the namespace owning the data type must include a data type stub declaration for the data type.

Data Type Members

The data type being defined is the current scope for all property definitions within it. As for a class (see 10.4.2), the properties of a data type define *attributes* that may take on values of the appropriate type in instances of the data type.

NOTE. Alf does not allow private or protected visibility to be specified for data type attributes. Since operations and nested classifiers are not allowed on data types, there would be no way to access data type attributes that are private or protected. Both public and package visibility are allowed.

A property definition may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing data type, rather than in model scope.

Data Type Specialization

A data type may *specialize* one or more other data types, in which case it *inherits* attributes from the data types it specializes (its *supertypes*). Each of the names in the specialization part of a data type must resolve to a data type. The visibility of inherited members from the data types named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier.

All non-private attributes of the supertypes are inheritable. However, all attributes of a data type, whether owned or inherited, must have unique names. It is therefore not legal to define a data type to inherit attributes with the same name as each other or any defined in the subtype.

NOTE. The constraint on distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. The default definition for distinguishability of named elements is given in 7.3.33 of the UML Superstructure.

A data type may be defined to be *abstract.* An abstract data type cannot be instantiated, but it can be used as a supertype of other data types.

Data Type Instantiation

Like an object, a data value is created as an instance of a data type using an instance creation expression (see 8.3.12). However, unlike objects, the data values do not have independent identity and two data values of the same type are considered equal if the values of their corresponding attributes are equal (see also 8.6.6 on the semantics of equality).

10.4.5 Associations

An *association* is a classifier that specifies a semantic relationship that may exist between two or more instances. The instances of an association are known as *links.* The features of an association must all be properties (see 10.5.2).

Examples

```
assoc Selection {
  public cart:             ShoppingCart[0..*]; // Association end definitions
  public selectedProduct:  Product[1..*];
  public selectionInfo:    ProductSelection;
}
```

Syntax

```
AssociationDeclaration(d: AssociationDefinition)
      = [ "abstract" (d.isAbstract=true) ] "assoc" ClassifierSignature(d)
AssociationDefinition(d: AssociationDefinition)
      = AssociationDeclaration(d) "{" StructuredMember(d.ownedMember)
        StructuredMember(d.ownedMember) { StructuredMember(d.ownedMember) }
        "}"
```

NOTE. An association must have at least two association ends.

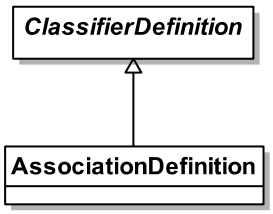

Figure 10.8 Abstract Syntax of Association Definitions

Cross References

1. ClassifierDefinition see 10.4.1
2. ClassifierSignature see 10.4.1
3. StructuredMember see 10.4.4

Semantics

If the association definition is a subunit definition, then the definition of the namespace owning the association must include an association stub declaration for the association.

Association Ends

The association being defined is the current scope for all property definitions within it. The properties of an association are the *association ends* whose values are the instances being related by a specific instance of the association.

NOTE. Per the fUML subset, association ends, as structural features, are always owned by their association (see fUML Specification, 7.2.2). Alf does not allow private or protected visibility to be specified for association ends, because there would be no way to access them. Public and package visibility are allowed.

A property definition may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing association, rather than in model scope.

If an association definition contains an association end that is a composed property, then it must have exactly two association ends.

NOTE. The UML Superstructure, 7.3.3, requires that a composition association be binary.

Association Specialization

An association may *specialize* one or more other associations, in which case it *inherits* association ends from the associations it specializes (its *superassociations*). Each of the names in the specialization part of an association must resolve to an association. The visibility of inherited association ends from the associations named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier.

All non-private association ends of superassociations are inheritable. However, all association ends of an association, whether owned or inherited, must have unique names. It is therefore not legal to define an association to inherit association ends with the same name as each other or any defined in the subassociation.

NOTE. The constraint on distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. The default definition for distinguishability of named elements is given in 7.3.33 of the UML Superstructure.

An association may be defined to be *abstract.* An abstract association cannot be instantiated, but it can be used as a superassociation of other associations.

Association Instantiation

Links are created and destroyed as instances of an association using the `createLink` and `destroyLink` link operations (see 8.3.13). Links are not themselves values. However, the properties of the links of an association may be queried using an association read expression (which has the form of a behavior invocation—see 8.3.9) or a property access expression (for a binary association—see 8.3.6).

10.4.6 Enumerations

An *enumeration* is a classifier whose allowed instances are a specified set of *enumeration literals.*

Examples

```
enum TrafficLightColor { RED, YELLOW, GREEN }
```

Syntax

```
EnumerationDeclaration(d: EnumerationDefinition)
      = "enum" Name(d.name) [ SpecializationClause(d.specialization) ]
EnumerationDefinition(d: EnumerationDefinition)
      = EnumerationDeclaration(d) "{" EnumerationLiteralName(d.ownedElement)
        { "," EnumerationLiteralName(d.ownedElement) } "}"
EnumerationLiteralName(m: EnumerationLiteralName)
      = [ DocumentationComment(m.documentation) ]
        Name(m.name) (m.visibility="public")
```

NOTE. Enumerations cannot have template parameters, since there would not be any way to use them within the enumeration definition.

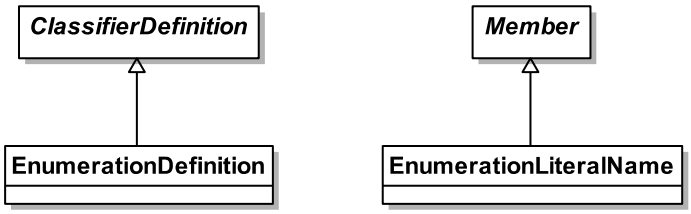

Figure 10.9 Abstract Syntax of Enumeration Definitions and Enumeration Literal Names

Cross References

1. DocumentationComment see 7.5.3
2. Name see 7.6
3. Member see 10.2
4. ClassifierDefinition see 10.4.1
5. SpecializationClause see 10.4.2

Semantics

If the enumeration definition is a subunit definition, then the definition of the namespace owning the enumeration must include an enumeration stub declaration for the enumeration.

Enumeration Literals

The enumeration being defined is the current scope for all enumeration literal names defined within it. Enumeration literals are the only members allowed for an enumeration.

NOTE. Alf assumes public visibility for all enumeration literals.

Enumerations are not actually instantiated, but, rather, their enumeration literals are simply referenced by name (see 8.3.3).

Enumeration Specialization

An enumeration may *specialize* one or more other enumerations, in which case it *inherits* enumeration literals from the enumerations it specializes (its *supertypes*). Each of the names in the specialization part of an enumeration must resolve to an enumeration. The visibility of inherited members from the enumerations named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier.

All enumeration literals of the supertypes are inheritable. However, all enumeration literals of an enumeration, whether owned or inherited, must have unique names. It is therefore not legal to define an enumeration to inherit members with the same name as each other or any defined in the subtype.

NOTE. The constraint on distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. The default definition for distinguishability of named elements is given in 7.3.33 of the UML Superstructure. Since an enumeration is not actively instantiated, there is no reason to be able to define one as abstract.

10.4.7 Signals

A *signal* is a classifier whose instances may be sent asynchronously (see 8.3.10) to an active object (see also 10.4.3). The ability for an object to receive a certain signal is specified using a *reception* declaration in the class of the object (see 10.5.4). The features of a signal must all be properties (see 10.5.2).

Examples

```
signal SubmitCharge {
    public accountNumber:      BankCardAccountNumber;
    public billingAddress:     MailingAddress;
    public cardExpirationDate: MonthYear;
    public cardholderName:     PersonalName;
}
```

Syntax

```
SignalDeclaration(d: SignalDefinition)
      = [ "abstract" (d.isAbstract=true) ] "signal" ClassifierSignature(d)
SignalDefinition(d: SignalDefinition)
      = SignalDeclaration(d) "{" { StructuredMember(d.ownedMember) } "}"
```

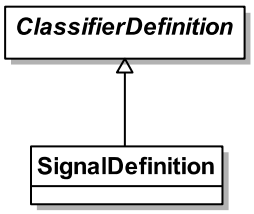

Figure 10.10 Abstract Syntax of Signal Definitions

Cross References

1. ClassifierDefinition see 10.4.1
2. ClassifierSignature see 10.4.1
3. StructuredMember see 10.4.4

Semantics

If the signal definition is a subunit definition, then the definition of the namespace owning the signal must include a signal stub declaration (or signal reception stub declaration—see 10.5.4) for the signal.

Signal Attributes

The signal being defined is the current scope for all property definitions within it. The properties of a signal are the *attributes* of the signal whose values in an instance of the signal are data that is transmitted by sending the signal.

NOTE. Alf does not allow private or protected visibility to be specified for attributes of a signal, because there would be no way to access them. Public and package visibility are allowed.

A property definition may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing signal, rather than in model scope.

Signal Specialization

A signal may *specialize* one or more other signals, in which case it *inherits* attributes from the signals it specializes (its *supersignals*). Each of the names in the specialization part of a signal must resolve to a signal. The visibility of inherited members from the signals named in the specialization part is as specified in the UML Superstructure, 7.3.8, Classifier.

All non-private attributes of the supersignals are inheritable. However, all attributes of a signal, whether owned or inherited, must have unique names. It is therefore not legal to define a signal to inherit attributes with the same name as each other or any defined in the subsignal.

NOTE. The constraint on distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. The default definition for distinguishability of named elements is given in 7.3.33 of the UML Superstructure.

A signal may be defined to be *abstract.* An abstract signal cannot itself be sent (see 8.3.10), but it can be used as a supersignal of other signals.

Signal Instantiation

Signals are implicitly instantiated as part of the asynchronous feature invocation of a reception (see 8.3.10). Such instances may be received using an accept statement (see 9.15). Once received, the attributes of a signal instance (if any) may be accessed just like the properties of any other kind of instance (see 8.3.6).

10.4.8 Activities

An *activity* is “the specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions” (UML Superstructure, 12.3.4). It is the fundamental mechanism for behavioral modeling in Alf.

Activities are namespaces (see 10.2) that may be defined as Alf units (see 10.1). Activities may also be implicitly defined by the in-line specification of behavior for active classes (see 10.4.3) or operations (see 10.5.3).

In UML, activities are also classes (see UML Superstructure, 12.3.4) and so may have attributes, operations and specialization. However, for simplicity, Alf does not provide a textual notation for features and specializations on activities. Instead, an active class may be used to model structural features along with a related behavior (see 10.4.3).

Examples

```
activity getNodeActivations
  (in node: ActivityNode): ActivityNodeActivation[*] {
  return this.activations -> select a (a.node == node);
}
```

```
activity execute()
{
  //@parallel
  {
    'activity' = (Activity)(this.types[1]);
    {
      group = new ActivityNodeActivationGroup();
      group.activityExecution = this;
    }
    {
      this.activationGroup = group;
      group.activate('activity'.node, 'activity'.edge);
    }
  }
}
```

```
activity Order_Behavior() {
  accept (checkout: Checkout);
  this.establishCustomer(checkout);

  do {

    accept (chargeSubmission: SubmitCharge);
    this.processCharge(chargeSubmission);

    accept (PaymentDeclined) {
      declined = true;
      this.declineCharge();

    } or accept (PaymentApproved) {
      declined = false;
    }

  } while (declined);

  this.packAndShip();
  accept (OrderDelivery);
  this.notifyOfDelivery();
}
```

Syntax

```
ActivityDeclaration(d: ActivityDefinition)
      = "activity" Name(d.name) [ TemplateParameters(d) ] FormalParameters(d)
        [ ReturnParameter(d.ownedMember) ]
ActivityDefinition(d: ActivityDefinition)
      = ActivityDeclaration(d) Block(d.body)
FormalParameters(d: NamespaceDefinition)
      = "(" [ FormalParameterList(d) ] ")"
FormalParameterList(d: NamespaceDefinition)
      = FormalParameter(d.ownedMember) { "," FormalParameter(d.ownedMember) }
FormalParameter(p: FormalParameter)
      = [ DocumentationComment(p.documentations) ]
        { StereotypeAnnotation(p.annotations) }
        ParameterDirection(p.direction) Name(p.name) ":"
        TypePart(p)
ParameterDirection(dir: String)
      = "in"(dir) | "out"(dir) | "inout"(dir)
```

```
ReturnParameter(p: FormalParameter)
      = ":" TypePart(p) (p.direction="return")
```

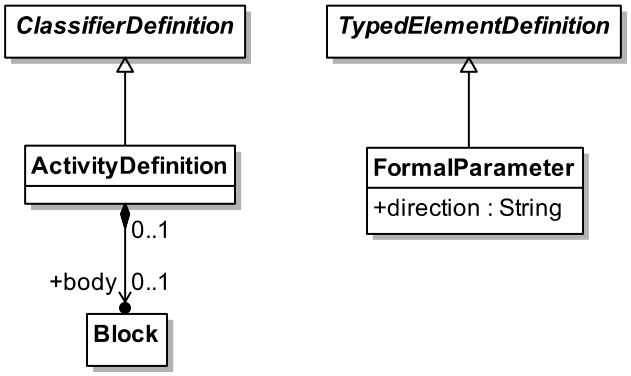

Figure 10.11 Abstract Syntax of Activity Definitions and Formal Parameters

Cross References

1. Block see 9.1
2. ClassifierDefinition see 10.4.1
3. TypePart see 10.5.2
4. TypedElementDefinition see 10.5.2

Semantics

Formal Parameters

The definition of an activity includes the names and types of any `in`, `out` and `inout` parameters, as well as, optionally, the type of a single return parameter. The activity acts as the namespace for its parameters (but parameters are not packageable elements and, therefore, cannot be imported; see 10.3). Return parameters cannot be named in Alf. The type part of a formal parameter definition statically specifies the type and multiplicity of the formal parameter in the same way as for a property (see 10.5.2).

A formal parameter may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing activity, rather than in model scope.

Activities as Units

An activity definition can either be a model unit in itself or a subunit of another Alf unit. As a subunit, an activity definition may complete a stub declaration for an activity as a nested classifier (see 10.4.2), as the classifier behavior of an active class (see 10.4.3) or as the method of an operation (see 10.5.3).

An activity definition that completes a classifier behavior declaration may not be a template and may not have any parameters. An activity definition that completes an operation also may not be a template but must have formal and return parameters (if any) that match exactly, in order, the parameters of the operation in direction, name, type and multiplicity. An activity definition that completes a nested activity stub must also match the parameters of the stub, as for an operation. (The type name of a parameter of an activity does not need to lexically match exactly that of the corresponding operation parameter, but

the activity parameter type name must resolve to the same classifier as the corresponding parameter of the operation, or the activity and operation parameters must both be untyped.)

Note also that any stereotype annotations made on an activity definition subunit (see 10.1) that completes an operation maps to a stereotype on the *method* for that operation, not the operation itself. Stereotype annotations for the operation itself (such as `@Create` for a constructor or `@Destroy` for a destructor) must be made on the operation stub declaration (see 10.4.2).

Local and Parameter Names

An activity can introduce *local names* for elements within it. Such named elements are *not* namespace elements of the activity. Local names are used in Alf to denote intermediate values in computations within an activity. The scope of such local names is generally from the point at which they are defined lexically to the end of the containing activity. However, the local names defined in an activity are never visible outside that activity. (See also the discussion of local names in subclauses 8.1 and 9.1.)

The names of the parameters of an activity are not technically local names, since they are owned namespace members of the activity. However, they are used within the body of an activity in much the same way as local names. Further, while parameter names may be qualified with the name of the activity, this is not required, meaning that they are generally written as unqualified names, again much like local names.

An `in` parameter may be referenced by name within the body of an activity (see 8.3.3), but it cannot be reassigned. It is treated similarly to a local name whose assigned source is always given by its input value at the start of the execution of an activity.

Both `out` and `inout` parameters, on the other hand, may be referenced and assigned (see 8.8) within the body of an activity. An `out` parameter is initially unassigned, while an `inout` parameter is initially assigned its input value at the start of the execution of an activity. The values returned for these parameters by the activity are from their final assigned sources. If an `out` parameter has a multiplicity lower bound greater than 0, then it must have an assigned source after the last statement in the block given in the activity definition.

Activity Execution

An activity may be executed as the classifier behavior of an active class (see 10.4.3), as an operation method (see 10.5.3) or as a stand-alone behavior in its own right. A classifier behavior is automatically started asynchronously when an instance of the class that owns it is created (see 8.3.12). A stand-alone activity can also be started asynchronously by instantiating it as a class, as long as it does not have parameters (see 8.3.12), or it can be called synchronously using a behavior invocation (see 8.3.9). An operation method can only be called synchronously (see 8.3.10).

However it is invoked, an activity is then executed by executing the block given in its definition (see 9.1). That is, each of the statements in the block are executed sequentially in order, and the activity terminates once the last statement in the block completes execution or if a `return` statement is executed (see 9.14). If the activity has a return parameter with a multiplicity lower bound greater than 0, then it must be statically determinable that the activity will terminate by the execution of a `return` statement.

10.5 Features

10.5.1 Overview

A *feature* declares a behavioral or structural characteristic of the instances of a classifier. A structural characteristic is declared using a *property* (see 10.5.2). A behavioral characteristic is declared using an *operation* (see 10.5.3) or (for an active class) a *reception* (see 10.5.4).

Syntax

```
FeatureDefinition(m: Member)
      = AttributeDefinition(m)
      | OperationDefinition(m)
FeatureStubDeclaration(m: Member)
      = OperationStubDeclaration(m)
ActiveFeatureDefinition(m: Member)
      = ReceptionDefinition(m)
      | SignalReceptionDefinition(m)
ActiveFeatureStubDeclaration(m: Member)
      = SignalReceptionStubDeclaration(m)
```

Cross References

1. Member see 10.2
2. AttributeDefinition see 10.5.2
3. OperationDefinition see 10.5.3
4. OperationStubDeclaration see 10.5.3
5. ConstructorDefinition see 10.5.3.1
6. ConstructorStubDeclaration see 10.5.3.1
7. DestructorDefinition see 10.5.3.2
8. DestructorStubDeclaration see 10.5.3.2
9. ReceptionDefinition see 10.5.4
10. SignalReceptionDefinition see 10.5.4
11. SignalReceptionStubDeclaration see 10.5.4

Semantics

See the discussion of the semantics of each kind of feature in subsequent subclauses.

10.5.2 Properties

A *property* is a structural feature of a classifier. The attributes of classes, data types and signals are properties, as are the association ends of an association.

Example

```
amount: Money = 0;
products: Product [1..*] ordered;
wheels: compose Wheel [2..4];
position: Point = new(0,0);
```

```
colors: Set<Color> = { Color::red, Color::blue, Color::green };
```

Syntax

```
PropertyDefinition(d: PropertyDefinition)
      = PropertyDeclaration(d) ";"
AttributeDefinition(d: PropertyDefinition)
      = PropertyDeclaration(d) [ AttributeInitializer(d.initializer) ] ";"
AttributeInitializer(e: Expression)
      = "=" InitializationExpression(e)
PropertyDeclaration(d: PropertyDefinition)
      = Name(d.name) ":" [ "compose" (d.isComposite=true) ] TypePart(d)
TypePart(d: TypedElementDefinition)
      = TypeName(d.typeName) [ Multiplicity(d) ]
Multiplicity(d: TypedElementDefinition)
      = MultiplicityRange(d) [ OrderingAndUniqueness(d) ]
OrderingAndUniqueness(d: TypedElementDefinition)
      = "ordered" (d.isOrdered=true) [ "nonunique" (d.isNonunique=true) ]
      | "nonunique" (d.isNonunique=true) [ "ordered" (d.isOrdered=true) ]
      | "sequence" (d.isOrdered=true and d.isNonunique=true)
MultiplicityRange(d: TypedElementDefinition)
      = MultiplicityIndicator (d.upperBound="*")
      | "[" [ DecimalLiteral(d.lowerBound) ".." ]
        UnlimitedNaturalLiteral(d.upperBound) "]"
UnlimitedNaturalLiteral(v: String)
      = DecimalLiteral(v)
      | UnboundedValueLiteral(v)
```

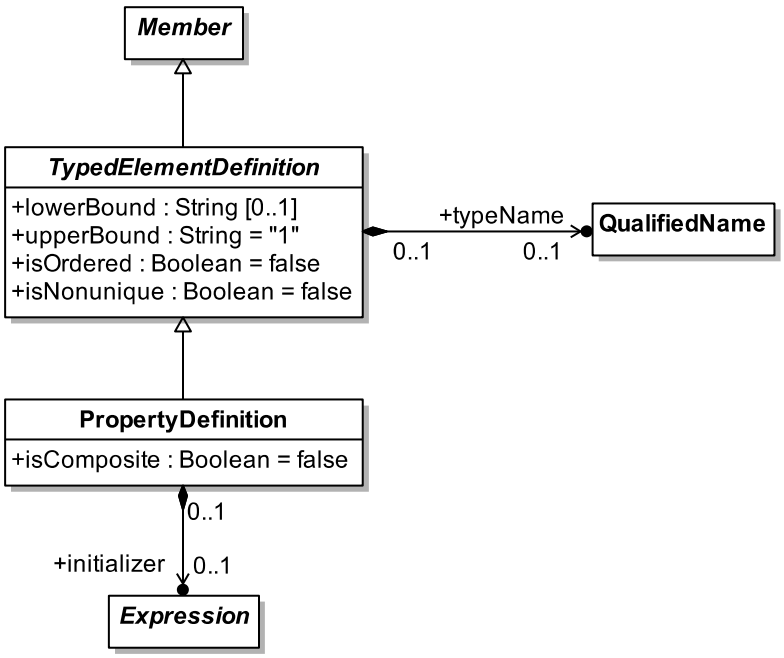

Figure 10.12 Abstract Syntax of Property Definitions

Cross References

1. Name see 7.6
2. NaturalLiteral see 7.8.3

3. UnboundedValueLiteral see 7.8.4
4. Expression see 8.1
5. QualifiedName see 8.2
6. TypeName see 8.2
7. InitializationExpression see 9.6
8. Member see 10.2

Semantics

Property Definition

The classifier that owns a property provides the current scope for naming in the property definition.

The property definition statically specifies the type and multiplicity associated with the property name. If the property type is given by a qualified name, then this name must resolve to a classifier. This classifier may not be a template, though it may be the binding of a template classifier with arguments provided for all template parameters.

Alternatively, a property may by *untyped.* An untyped property is indicated by using the keyword `any` in place of a classifier name.

The type of a property restricts the values that may be held by the property to instances of the given type. If the property is untyped, there are no restrictions on the value the property may hold.

The multiplicity of a property specifies the upper and lower bounds of the cardinality of values a property may have in any one instance of its owning classifier. If no multiplicity is specified for a property, the default multiplicity is `[1..1]`. If only a single bound is specified, then this is considered to be both the upper and lower bound, except in the case of a multiplicity specified as `[*]`, which is equivalent to `[0..*]`. A multiplicity specification of `[]` (that is, brackets with no explicit bounds) is also considered equivalent to `[0..*]`.

By default, a property with a multiplicity upper bound greater than 1 is considered to be unordered and non-unique. However, this default may be overridden by using the keywords `ordered` and/or `nonunique` in the property definition.

The Alf notation for properties is thus similar to the usual notation used in UML diagrams. For example, the property definition

```
items: Item [0..*] ordered nonunique;
```

declares a property that can hold zero or more objects of type `Item` in an ordered sequence. If the multiplicity is `[0..*]`, then a shorter form can also be used in which the range `0..*` is implicit:

```
items: Item[] ordered nonunique;
```

In addition, the single keyword "`sequence`" may be used in place of the combination of the two keywords "`ordered`" and "`nonunique`". Thus, the following is also equivalent to the declarations above:

```
items: Item[] sequence;
```

Composition

If a property definition includes the keyword "`compose`", then it is considered to be a *composition* of the values it holds. This has semantic implications when an instance of the owning classifier of the property is destroyed (see 10.5.3.2).

For example, consider the class

```
class C {
  public a: A = new A();
  public b: compose B = new B();
}
```

When an instance of class `C` is destroyed, the object it holds for attribute `b` will also be automatically destroyed, but the object it holds for attribute `a` will *not.*

Composition properties can also be included in association definitions. For example:

```
assoc R {
  public c: C;
  public d: compose D;
}
```

Note that (per UML Superstructure, 7.3.3) the composition annotation is on the *part* end of the composite association. That is, in the above association, `c` is the *composite* while `d` is the *part.* Thus, when an instance of class `C` is destroyed, if there is a link of association `R` with that object at one end, then that link *and* the instance of `D` at the other end will also be destroyed.

NOTE. Alf provides no notation for shared aggregation, since this has no semantic effect. However, properties specified outside of Alf notation with aggregation=shared are treated the same way as properties with aggregation=none.

Attribute Initialization

The attributes of a class are properties (see 10.4.2). However, unlike the case of other property definitions, Alf provides a notation for the *initializer* of an attribute, which is an expression that is evaluated every time the class containing the attribute is instantiated, with the result being assigned to the attribute. The enclosing class is the current scope for names in the expression. The evaluation of initializers is carried out as part of the execution of the creation of a newly instantiated object (see 8.3.12 on the creation of objects and 10.5.3.1 on constructors).

NOTE. The fUML subset does not include default values for properties (see fUML Specification, 7.2.2). However, the evaluation of attribute initializer expressions in Alf are mapped as part of the constructor for the enclosing class that owns the attribute, which can be executed within the fUML subset. The attribute initializers themselves do not need to be referenced during model execution.

Instance creation (see 8.3.12) and sequence construction (see 8.3.15) expressions use as initializers may be written in the shorthand *initialization expression* form, as for a local name declaration statement (see 9.6). For example, the attribute definition

```
position: Point = new(0,0);
```

is equivalent to

```
position: Point = new Point(0,0);
```

and the attribute definition

```
colors: Set<Color> = { Color::red, Color::blue, Color::green };
```

is equivalent to

```
colors: Set<Color> = Set<Color>{ Color::red, Color::blue, Color::green };
```

10.5.3 Operations

An *operation* is a behavioral feature of a class that provides the specification for invoking an associated *method* behavior. Only classes may have operations as features. An operation is called on an instance of a class that has it as a feature using an invocation expression (see 8.3.10).

Examples

```
// Abstract operation
abstract select(in cart: Cart, in product: Product, in quantity: Count);

// In-line definition
select(in cart: Cart, in product: Product, in quantity: Count) {
  Selects.createLink(cart, product, this);
  this.quantity = quantity;
  this.unitPriceOfSelection = product.getUnitPrice();
}

// Stub declaration
unitPrice(): Money redefines Selection::getUnitPriceOfSelection;
```

Syntax

```
OperationDeclaration(d: OperationDefinition)
      = [ "abstract" (d.isAbstract=true) ] Name(d.name)
        FormalParameters(d.ownedMember) [ ReturnParameter(d.ownedMember) ]
        [ RedefinitionClause(d.redefinition) ]
OperationDefinition(d: OperationDefinition)
      = OperationDeclaration(d) Block(d.body)
OperationStubDeclaration(d: OperationDefinition)
      = OperationDeclaration(d) ";" (d.isStub=not d.isAbstract)
RedefinitionClause(qList: QualifiedNameList)
      = "redefines" QualifiedNameList(qList)
```

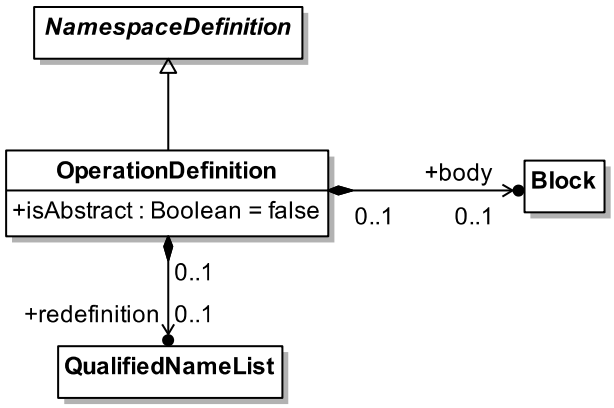

Figure 10.13 Abstract Syntax of Operation Definitions

Cross References

1. Name see 7.6
2. Block see 9.1
3. QualifiedNameList see 9.15
4. NamespaceDefinition see 10.2

5. FormalParameters see 10.4.8
6. ReturnParameter see 10.4.8
7. TypePart see 10.5.2

Semantics

Operation Signature

The definition of an operation includes the names and types of any `in`, `out` and `inout` parameters, as well as, optionally, the type of a single return parameter. The operation acts as the namespace for its parameters (but parameters are not packageable elements and, therefore, cannot be imported; see 10.3). Return parameters cannot be named in Alf. The type part of a formal parameter definition statically specifies the type and multiplicity of the formal parameter in the same way as for a property (see 10.5.2).

The *signature* is the list of types of the parameters of an operation, in the order of the parameters. If an operation has a return parameter, it is always listed after any other parameters. The directions (`in`, `out`, `inout`, `return`) of the parameters are *not* considered part of the operation signature.

A formal parameter may also have one or more stereotype annotations applied to it. Such annotations have the same semantics as annotations made on a unit definition (see 10.1), except that the qualified names do not need to be fully qualified and are resolved in the current scope of the enclosing activity, rather than in model scope.

Operation Distinguishability

Two operations are considered *distinguishable* within the same namespace if they have different names or different signatures. It is illegal to define two indistinguishable operations within the same class.

For example:

```
class C {
  f (in x: Integer): Integer;
  f (in x: Boolean): Integer;   // Legal, different parameter type.
  f (in x: Integer): Boolean;   // Legal, different return type.
  f (out x: Integer): Integer;  // Illegal! Parameter types are the same.
}
```

The first three operations above are distinguishable, even though they have the same name. They are said to be *overloaded.* The last operation, however, has the same signature as the first one, and is therefore illegal in the same class.

NOTE. The rule for distinguishability of operations is defined in general for behavioral features in 7.3.5 of the UML Superstructure. The requirement for distinguishability of namespace members is given in 7.3.34 of the UML Superstructure.

Operation Redefinition

It is also illegal to inherit an operation that is not distinguishable from all other owned and inherited operations of a class (see also 10.4.2). However, an operation from a superclass that is *redefined* in a subclass is not considered to be inherited into the subclass. It is therefore legal for the redefining operation to have the same name and signature as the operation it redefines.

In Alf, if an operation is defined in a subclass that is indistinguishable from an operation that would otherwise be inherited from a superclass, then the subclass operation is instead considered to redefine

the superclass operation. If there are multiple indistinguishable operations inheritable from different superclasses, then the redefining operation is considered to redefine them all.

It is also possible to explicitly specify an operation redefinition. In this case, the name of the redefining operation may be different than the name of the redefined operation. Each qualified name in the redefinition clause in the declaration of the redefining operation must resolve to a visible operation of a superclass that is consistent with the redefining operation being declared, as specified in UML Superstructure, 7.3.36, Operation.

For example, in the following class definitions:

```
class A { p(in x: Integer); }

class B ( p(in x: Integer); }

class C specializes A,B { p(in x: Integer); }
```

the final class definition is equivalent to

```
class C specializes A,B {
  p(in x: Integer) redefines A::p, B::p;
}
```

Note that, in this case, it would be illegal to redefine only one of `A::p` and `B::p` with an indistinguishable operation, since then the other superclass operation would still be inherited and would conflict with the redefining operation in `C`.

In the case of the explicit definition, it would also be possible to give the operation in `C` a different name:

```
class C specializes A,B {
  renamed(in x: Integer) redefines A::p, B::p;
}
```

Note that, in this case, the operations `A::p` and `B::p` are still *not* inherited by `C`—`C` has no operation called `p`. On the other hand, in

```
class C specializes A,B {
  renamed(in x: Integer) redefines A::p;
}
```

only `A::p` is redefined in `C`, as `renamed`, and `B::p` is, therefore, still inherited. But there is no name conflict, and `C` will have both an operation called `renamed` and an inherited operation called `p`.

NOTE. The rule for consistency between redefined and redefining operations is given in 7.3.36 of the UML Superstructure. The general UML rule is that the types of the redefining operation must conform to the types of the redefined operation. Unfortunately, this does not take into account the directions of the parameters. Alf requires that the redefining operation have parameters with the same types *and* directions as the redefined operation.

Operation Method

An operation is called using an invocation expression (see 8.3.10). The behavior of an operation is given by its *method.*

The method for an operation is specified as an activity that is a private owned behavior of the class (note that the *method* visibility is separate from the *operation* visibility). The operation may have a stub declaration, in which case its method must be separately defined as an activity subunit of the class owning the operation. Alternatively, the method may be specified using a block (see 9.1) in-line with the operation definition. The names of parameters defined in the operation declaration are visible (as local

names) within all constituent parts of the block in this case. As for an activity definition (see 10.4.8), if an `out` parameter has a multiplicity lower bound greater than 0, there must be an assigned source for it after the last statement of the block, and, if the operation has a return parameter with multiplicity lower bound greater than 0, then it must be statically determinable that execution of the block will terminate via a `return` statement.

The method of a redefining operation in a subclass *overrides* the methods of the operations it redefines. That is, when any of the redefined superclass operations are invoked on an instance of the subclass, it is the overriding method in the subclass that is executed, not the superclass method (see 8.3.10).

An *abstract* operation does not have a method. However, such an operation may be redefined in a subclass by a *concrete* operation with a method. Only abstract classes may have abstract operations (see 10.4.2). A concrete class that has superclasses with abstract operations must redefine all those operations to be concrete.

10.5.3.1 Constructors

A *constructor* is an operation, specially identified using the `@Create` stereotype annotation (see 10.1), and used to initialize a newly created object. The constructor to be used for this purpose is given in the instance creation expression that creates the object (see 8.3.12).

NOTE. Constructor operations are discussed in the UML Superstructure specification primarily in the context of composite structure (see UML Superstructure, 9.3.1). However, the «`Create`» stereotype is generally available to annotate a constructor operation (see UML Superstructure, C.1). An Alf constructor maps to an operation with this stereotype. Strictly, the use of this stereotype is outside the fUML subset. However, the stereotype is only used to signal that the annotated operation may be used in an instance creation expression. The actual mapping for an instance creation expression results in a create object action with a regular operation call on the constructor, which may be executed within the fUML subset (see 8.3.12).

Examples

```
@Create public Table(in rows: Integer, in columns: Integer);
```

```
@Create
public ProductSelection
  (in cart: Cart, in product: Product, in quantity: Count);
```

```
@Create
private registered() {
  Repository::get().register(this);
}
```

Semantics

Default Constructors

Every class represented in Alf notation (see 10.4.2) is always mapped as having at least one constructor. If no constructor is explicitly defined for the class, then the class is assumed to have a *default* constructor. The name of the default constructor is the same as the name of the class.

NOTE. This means that the instantiation of a class defined using Alf textual notation is never "constructorless" as defined in 8.3.12.

The behavior of the default constructor is to initialize any attributes owned by the class of a newly created object that have initializers in their definition (see 10.5.2). Such initialization has the semantics

of an assignment of the expression to the attribute (see 8.8). Attributes are initialized in the order in which they are defined in the class.

Note that an attribute initializer expression may refer to attributes within the same class, including the attribute being initialized. This means that the attribute being initialized and attributes defined later than the attribute being initialized may be used before they are themselves initialized. Such uninitialized attributes will be empty, which may violate the multiplicity lower bounds given in their definitions.

For example, consider the following class definition.

```
class Initialization {
  public a: Integer = this.b -> size();
  public b: Integer = 1;
}
```

An object created with the expression `new Initialization()` has `a` with value 0 (since `b` has not been initialized yet) and `b` with value 1.

Explicit Constructors

An explicit constructor definition has the same syntax as a regular operation definition, with the stereotype annotation `@Create`. An explicit constructor may have the name of its class, like a default constructor. However, it may also have a different name than that of its class, in which case the constructor is explicitly identified by name when used in an instance creation expression (see 8.3.12).

If a class has an explicit constructor definition, then the default constructor is no longer available, even if the defined constructor has a different name than the default constructor.

A constructor may also have parameters, but no return type is explicitly given. Implicitly, every constructor has the class it is constructing as its return type. This implicit return type is included in the signature for the constructor.

A class may have more than one constructor, any of which may be used in an instance creation expression for objects of the class.

As operations, non-private constructors are inheritable and the usual distinguishability rules apply (see 10.5.3). Note that constructors from different classes are always distinguishable, though, because at least their return types will be different.

Unlike a normal operation, a constructor may not be redefined in a subclass. Only a constructor directly owned by a class may be used in an instance creation expression for an instance of the class (see 8.3.12).

When an object is initialized using an explicit constructor, the default constructor attribute initialization behavior (as described above) is always performed before the explicit constructor behavior. For example, suppose the above example were modified as shown below.

```
class Initialization {
  public a: Integer;
  public b: Integer = 1;

  @Create public Initialization() {
    this.a = this.b -> size();
  }
}
```

The creation expression `new Initialization()` now results in an object having `a` with value 1, since attribute `b` is initialized before the body of the constructor is executed.

The body of a constructor may contain an *alternative constructor invocation* for another constructor in the same class or *super constructor invocations* for constructors in immediate superclasses. The syntax for such invocations is the same as a normal operation invocation (sees 8.3.10 and 8.3.11). In addition, only within a constructor body, the symbol "`this`" may be used as an alternative constructor invocation target, with the same meaning as invoking a constructor with the same name as the class. If the class has exactly one superclass, then the symbol "`super`" may also be used as a super constructor invocation target, with the same meaning as invoking a constructor on the superclass with the same name as the superclass.

For example, the following class definition contains an explicit invocation of one of its own constructors and one from its superclass.

```
class B specializes A {
  @Create public B(in x: Integer) {
    super.A(x);
  }

  @Create public B() {
    this.B(x);
  }
}
```

Using the special invocation syntax allowed in constructors, this could also be written as follows.

```
class B specializes A {
  @Create public B(in x: Integer) {
    super(x);
  }

  @Create public B() {
    this(x);
  }
}
```

An alternative constructor invocation may only occur as the first statement of the body of a constructor (as specified in the static semantics for feature invocation expressions in 8.3.10). Super constructor invocations must all occur at the beginning of the body of a constructor, with no other statements preceding them, and no more than one invocation for each superclass (as specified in the static semantics for super invocation expressions in 8.3.11).

In the absence of explicit constructor invocations at the start of a constructor body (and also in the case of the default constructor behavior), a `super` constructor invocation is made implicitly for each immediate superclass, in the order the superclasses appear in the `specializes` list of the class containing the constructor, before executing any statements in the constructor body. If the constructor body begins with explicit superclass constructor invocations for some but not all superclasses of the class containing the constructor, then `super` constructor invocations are made implicitly for all remaining superclass, before executing any statements in the constructor body.

If a class has multiple superclasses, then it is possible that these superclasses may themselves have one or more common ancestor classes (this is sometimes referred to as "diamond inheritance"). In this case, the above rules may result in the same constructor of a common ancestor class being called more than once or more than one constructor from the same common ancestor class being called. However, once a constructor from a class is called on an object, that object is considered to be *initialized for* that class. If another call is made, either explicitly or implicitly, on an object using any constructor from a class for

which the object is already initialized, then that constructor has no further effect: no default initialization is carried out, no implicit super constructor calls are made and the body of the constructor is not executed.

10.5.3.2 Destructors

A *destructor* is an operation, specially identified using the `@Destroy` stereotype annotation (see 10.1), used to clean up an object that is to be destroyed. A call to a destructor not only invokes the destructor's behavior, but also results in the actual destruction of the object (see the semantics of destructor invocation under subclauses 8.3.10 and 8.3.11).

NOTE. The UML Superstructure specification defines the «`Destroy`» stereotype to annotate a destructor operation (see C.1). An Alf destructor maps to an operation with this stereotype. Strictly, the use of this stereotype is outside the fUML subset. However, the stereotype is only used to signal that the annotated operation may be used in an instance destruction expression. The actual mapping for an instance destruction expression results in a regular operation call on the destructor, which may be executed within the fUML subset, followed by a destroy object action (sees 8.3.10 and 8.3.11).

Examples

```
@Destroy public ProductSelection();
```

```
@Destroy
private unregister() {
  Repository::get().unregister(this);
  super.destroy();
}
```

Semantics

Default Destructors

Every class represented in Alf notation (see 10.4.2) is always mapped as having at least one destructor. If no destructor is explicitly defined for the class, then the class is assumed to have a *default* constructor. The name of the default destructor is "`destroy`".

NOTE. This means that a call to the default destructor `destroy` is always an explicit destructor call, not an implicit destructor invocation as defined in 8.3.10.

The behavior of the default destructor is to first call the destructor `destroy()` (i.e., with no arguments) on any immediate superclasses (if such exists), in the order in which those superclasses are given in the `specializes` list for the class of the object being destroyed. The `destroy()` destructor (if it exists) is then called on any object that is the value of a composite attribute or on the opposite end of a composite association. Destructors are called on attributes in the order in which they are defined in the class of the object being destroyed. The order in which destructors are called on objects related by composite association is not specified.

Explicit Destructors

An explicit destructor definition has the same syntax as a regular operation definition, with the stereotype `@Destroy`. The name of the destructor may then be used in an invocation expression to destroy an object of the class (see 8.3.10).

If a class has an explicit destructor definition, then the default destructor is no longer available, even if the defined destructor has a different name than `destroy`.

A destructor may also have parameters, but may not have a return parameter.

A class may have more than one destructor, any of which may be used in an instance destruction expression for objects of the class.

As operations, non-private destructors are inheritable and the usual distinguishability rules apply (see 10.5.3). If a class has a destructor named `destroy`, with no arguments (either implicitly as the default destructor or explicitly), then a similar destructor named `destroy` with no arguments in a subclass implicitly redefines the superclass destructor. Otherwise, a destructor may not be explicitly redefined in a subclass. Only a destructor directly owned by a class may be used in an invocation expression used to destroy an instance of the class (see 8.3.10).

When an object is destroyed using an explicit destructor, the default destructor behavior is *not* performed, so any desired calls to superclass or composite part destructors must be made explicitly.

The body of a destructor may contain explicit invocations of other destructors in the same class (targeted to "`this`") or a superclass (targeted to "`super`"). However, such invocations act just like normal operation calls and do not cause the destruction of the object. An object is not actually destroyed until the completion of the original destructor invocation (sees 8.3.10 and 8.3.11).

For example, the following class definition contains an explicit invocation of one of its own destructors and of a destructor from its superclass.

```
class D specializes C {
  @Destroy public destroy() {
    super.destroy();
  }

  @Destroy public cancel (in reason: String) {
    WriteLine(reason);
    this.destroy();
  }
}
```

If `d` is an instance of `D`, then the expression `d.cancel("Example")` will result in the destruction of `d` only after the completion of the invocation of `cancel("Example")`, not by any of the intermediate destructor calls.

10.5.4 Receptions

A *reception* is a behavioral feature of an active class that declares that instances of the class are prepared to react to the receipt of a specific signal. Only active classes may have receptions as features. Normally, the signal is defined separately and referenced by name in the reception declaration. As a convenience, Alf also allows the definition of the signal and a declaration of a reception of it to be combined into a *signal reception definition.*

Examples

```
receive Checkout;
```

```
receive signal SubmitCharge {
  public accountNumber:      BankCardAccountNumber;
  public billingAddress:     MailingAddress;
  public cardExpirationDate: MonthYear;
  public cardholderName:     PersonalName;
}
```

Syntax

```
ReceptionDefinition(d: ReceptionDefinition)
      = "receive" QualifiedName(d.signalName) ";"
        (d.name=d.signal.nameBinding->last().name))
SignalReceptionDeclaration(d: SignalReceptionDefinition)
      = "receive" "signal" Name(d.name)
        [ SpecializationClause(d.specialization) ]
SignalReceptionDefinition(d: SignalReceptionDefinition)
      = SignalReceptionDeclaration(d) "{"
        { StructuredMember(d.ownedMember) } "}"
SignalReceptionStubDeclaration(d: SignalReceptionDefinition)
      = SignalReceptionDeclaration(d) ";" (d.isStub=true)
```

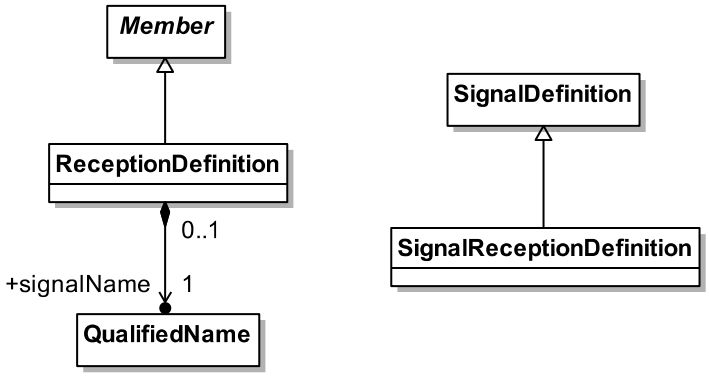

Figure 10.14 Abstract Syntax of Reception Definitions and Signal Reception Definitions

Cross References

1.	Name	see 7.6
2.	QualifiedName	see 8.2
3.	Member	see 10.2
4.	SpecializationClause	see 10.4.2
5.	StructuredMember	see 10.4.4
6.	SignalDefinition	see 10.4.7

Semantics

The owning classifier definition of a reception definition or a signal reception definition must be for an active class. An invocation of a reception on an instance of the owning active class results in a sending of the specified signal to that instance (see 8.3.10).

Reception Definitions

The name in a reception definition must be the visible name of a signal.

The reception is given the same name as the base name of the signal without any qualification. Since receptions must be distinguished by name, this means that no active class may have more than one reception (inherited or owned) for a signal with a given name.

NOTE. The general requirement for distinguishability of namespace members is given in 7.3.34 of the UML Superstructure. This is not overridden in 13.2.23 of the UML Superstructure on receptions.

Signal Reception Definitions

A signal reception definition defines both a reception as an owned feature of the active class and a signal as a nested classifier of the owning class of the reception. The static semantics for a signal definition (see 10.4.7) thus also apply to a signal reception definition. However, a signal reception definition may not have template parameters.

For example, the following active class definition:

```
active class Order {

  receive signal SubmitCharge {
    public accountNumber:      BankCardAccountNumber;
    public billingAddress:     MailingAddress;
    public cardExpirationDate: MonthYear;
    public cardholderName:     PersonalName;
  }

}
```

is equivalent to:

```
active class Order {

  receive SubmitCharge;

  signal SubmitCharge {
    public accountNumber:      BankCardAccountNumber;
    public billingAddress:     MailingAddress;
    public cardExpirationDate: MonthYear;
    public cardholderName:     PersonalName;
  }

}
```

Note that, even though the reception and the signal have the same name, there is no distinguishability conflict, because receptions and signals are separate syntactic types. The name `Order::SubmitCharge` in this example can thus refer to either the reception or the signal, depending on context.

11 Standard Model Library

11.1 Overview

Every modeling environment using Alf must also make available a set of models known as the *Alf Standard Model Library.* This library consists of the profile `ActionLanguage` and the model library package `Alf::Library`. The `Alf::Library` package contains the subpackages shown in Figure 11.1.

NOTE. The `ActionLanguage` profile is not nested within `Alf::Library` because it may also be useful outside the context of Alf.

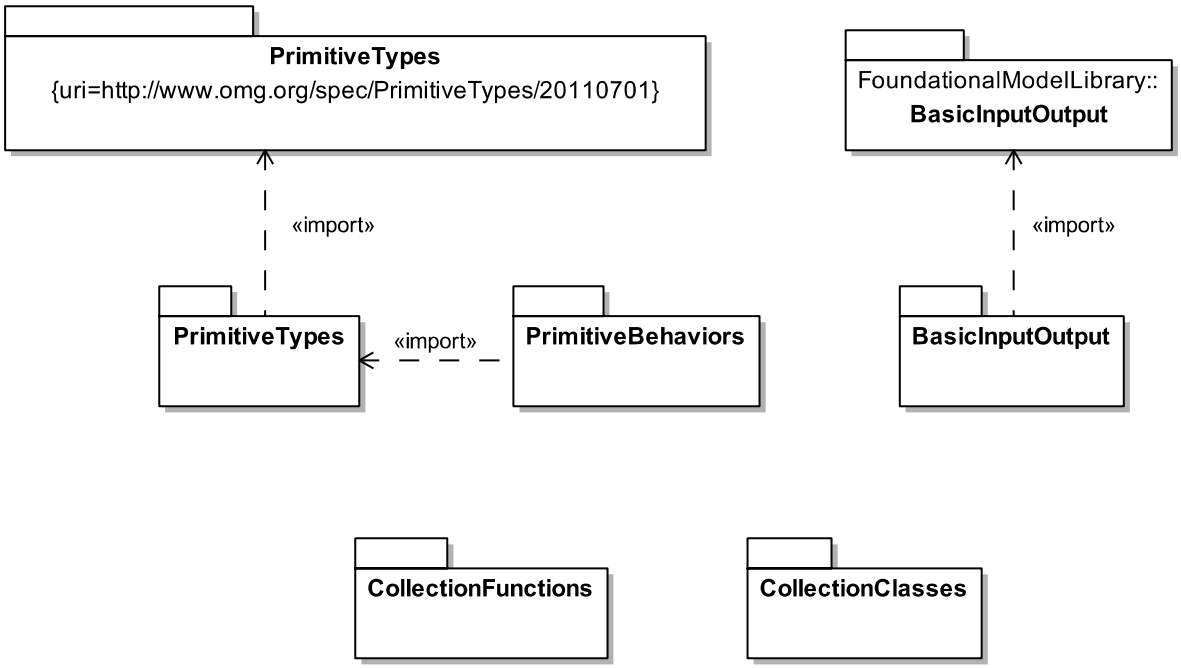

Figure 11.1 Alf Standard Model Library Subpackages

The `ActionLanguage` profile is described in 11.2. The subpackages of `Alf::Library` are then described in subsequent subclauses.

11.2 ActionLanguage Profile

The `ActionLanguage` profile contains the single stereotype shown in Figure 11.2. When applied to a comment, this stereotype indicates that the body of the comment is the textual representation of the element to which the comment is attached, potentially including the representation of other modeling elements contained in the element to which the comment is attached (see 6.2). The language attribute gives the name of the language in which the comment body text is written, using the same naming conventions as for the language attribute of an opaque behavior.

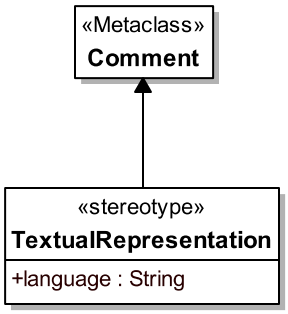

Figure 11.2 Stereotype `TextualRepresentation`

11.3 Primitive Types

The fUML Foundational Model Library imports the `PrimitiveTypes` package from the UML Infrastructure (see fUML Specification, Clause 9). Since the Alf Standard Model Library builds on the primitive behaviors defined in the fUML Foundational Model Library (see 11.4 below), the `Alf::Library::PrimitiveTypes` package also imports the UML `PrimitiveTypes` package, as shown in Figure 11.1 (see also UML Infrastructure, 13.1). This makes the UML primitive types `Integer`, `Boolean`, `String` and `UnlimitedNatural` available in the `Alf::Library::PrimitiveTypes` namespace. In addition to the imported types, the Alf `PrimitiveTypes` package also includes `Natural` and `BitString` types, as described in the remainder of this subclause.

To be recognized as primitive types in an fUML execution environment, the types defined in the `Alf::Library::PrimitiveTypes` package (including imported types) must be registered with any execution locus at which they are to be used (see fUML Specification, 8.2.1).

NOTE. While the `Real` primitive type defined in UML 2.4.1 is imported into the Alf `PrimitiveTypes` package, Alf does not currently provide operations or primitive behaviors to support this type. It is expected that such support will be provided in a subsequent version of Alf.

11.3.1 Natural Type

As shown in Figure 11.3, the primitive type `Natural` specializes both `Integer` and `UnlimitedNatural`. `Natural` literals have the type `Natural` unless they can be determined statically to be of type `Integer` or `UnlimitedNatural` from their context of use (see 7.8.3).

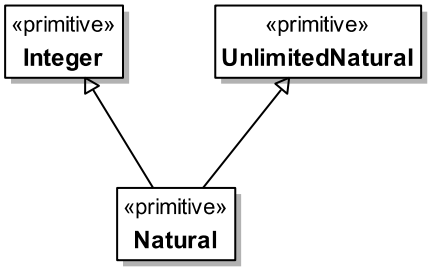

Figure 11.3 Primitive Type `Natural`

11.3.2 Bit String Type

The primitive type `BitString` represents values that are uninterpreted sequences of bits, each of which has either be *set* (bit value of 1) or *unset* (bit value of 0). Alf does not provide direct literals for bit strings, but the standard `BitStringFunctions` package provides functions for converting integers to bit strings and vice versa.

The *length* of a bit string is the number of bits in the string. For any conforming implementation, all bit strings must have the same length, but this length may differ between implementations. However, if an implementation limits supported integer values to a finite set (as permitted in 9.2 of the fUML Specification), the implemented bit string length must be at least long enough to represent the conversion from every supported integer value (positive or negative). If an implementation does not limit supported integer values, then the implemented bit string length must not be smaller than 32.

11.4 Primitive Behaviors

As shown in Figure 11.4, the `Alf::Library::PrimitiveBehaviors` package has subpackages corresponding to each of the subpackages in the fUML standard `FoundationalModelLibrary::PrimitiveBehaviors` package (see fUML Specification, 9.3) *except* for the `ListFunctions` package. Each of these primitive behavior packages imports the primitive functions from the corresponding fUML package, renaming certain functions with aliases so that they have names consistent with the corresponding operators used in the Alf expression syntax. In addition, the Alf `PrimitiveBehaviors` package also includes the `BitStringFunctions` and `SequenceFunctions` packages.

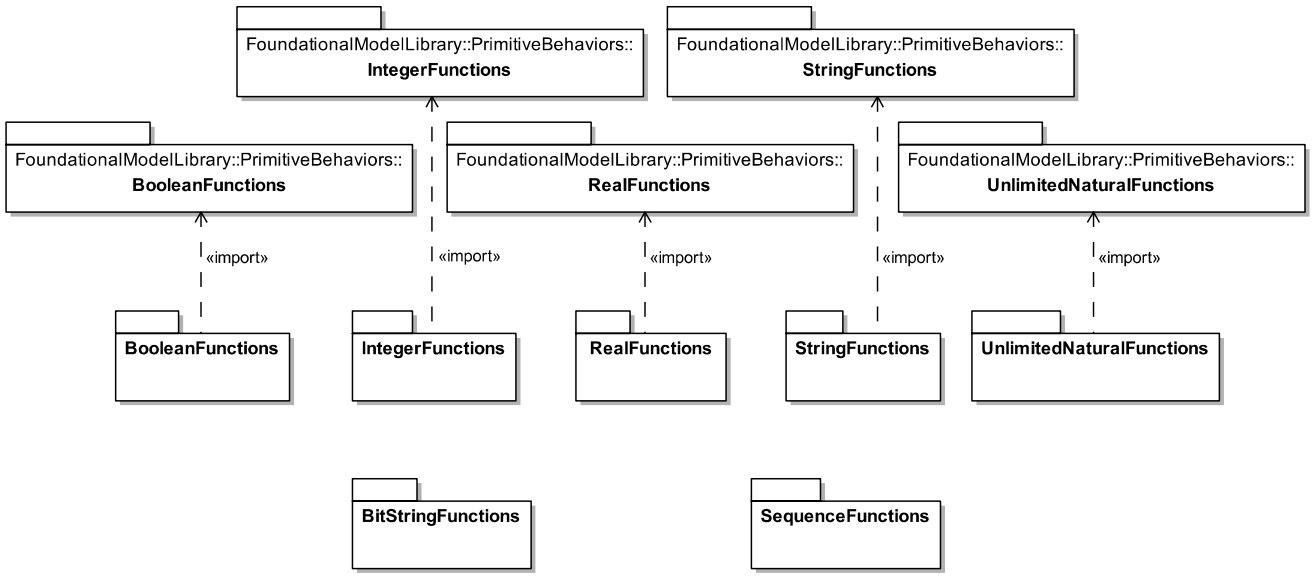

Figure 11.4 Primitive Behavior Packages

To be usable in an fUML execution environment, implementations of all of the function behaviors in each of the subpackages of the `Alf::Library::PrimitiveBehaviors` package (including imported behaviors) must be registered with any execution locus at which they are to be used (see fUML Specification, 8.2.1). These implementations must conform to the specifications for the behaviors they implement, as given in fUML Specification, 9.2, or in this subclause, as appropriate.

11.4.1 Boolean Functions

The `Alf::Library::PrimitiveBehaviors::BooleanFunctions` package imports all the behaviors contained in the fUML `FoundationalModelLibrary::PrimitiveBehaviors:: BooleanFunctions` package. Table 11.1 lists these functions with their Alf aliases, along with their corresponding fUML name. The formal specification for these functions is given in 9.3.1 of the fUML Specification.

Table 11.1 `Boolean` Functions

Function Signature	fUML Name	Description
`'|'(in x: Boolean, in y: Boolean): Boolean`	`Or`	True if either `x` or `y` is true.
`'^'(in x: Boolean, in y: Boolean): Boolean`	`Xor`	True if either `x` or `y` is true, but not both.
`'&'(in x: Boolean, in y: Boolean): Boolean`	`And`	True if both `x` and `y` are true.
`'!'(in x: Boolean): Boolean`	`Not`	True if `x` is false.
`Implies (in x: Boolean, in y: Boolean): Boolean`	`Implies`	True if `x` is false, or if `x` is true and `y` is true.
`ToString(in x: Boolean): String`	`ToString`	Converts `x` to a `String` value.
`ToBoolean(in x: String): Boolean[0..1]`	`ToBoolean`	Converts `x` to a `Boolean` value.

11.4.2 Integer Functions

The `Alf::Library::PrimitiveBehaviors::IntegerFunctions` package imports all the behaviors contained in the fUML `FoundationalModelLibrary::PrimitiveBehaviors::IntegerFunctions` package. Table 11.2 lists these functions with their Alf aliases, along with their corresponding fUML name. The formal specification for these functions is given in 9.3.2 of the fUML Specification.

The package also includes the additional `ToNatural` and `ToReal` functions, in addition to the functions imported from the fUML `IntegerFunctions` package. The `ToNatural` function converts a string representation of a natural (unsigned) number in any of the forms given in 7.8.3, as opposed to the `ToInteger` function which only converts from a decimal string representation (and also allows a sign). The `ToReal` function converts an integer value to a real value, equivalent to the result of using the fUML `IntegerFunctions::'/'` function (real division) to divide the integer value by 1.

Table 11.2 `Integer` Functions

Function Signature	fUML Name	Description
`Neg(in x: Integer): Integer`	`Neg`	The negative value of `x`.
`Abs(in x: Integer): Integer`	`Abs`	The absolute value of `x`.
`'+'(in x: Integer, in y: Integer): Integer`	`+`	The value of the addition of `x` and `y`.
`'-'(in x: Integer, in y: Integer): Integer`	`-`	The value of the subtraction of `x` and `y`.
`'*'(in x: Integer, in y: Integer): Integer`	`*`	The value of the multiplication of `x` and `y`.
`'/'(in x: Integer, in y: Integer): Integer[0..1]`	`Div`	The number of times that `y` fits completely within `x`.
`'%'(in x: Integer, in y: Integer): Integer`	`Mod`	The result is `x` modulo `y`.
`Max(in x: Integer, in y: Integer): Integer`	`Max`	The maximum of `x` and `y`.
`Min(in x: Integer, in y: Integer): Integer`	`Min`	The minimum of `x` and `y`.
`'<'(in x: Integer, in y: Integer): Boolean`	`<`	True if `x` is less than `y`.
`'>'(in x: Integer, in y: Integer): Boolean`	`>`	True if `x` is greater than `y`.
`'<='(in x: Integer, in y: Integer): Boolean`	`<=`	True if `x` is less than or equal to `y`.
`'>='(in x: Integer, in y: Integer): Boolean`	`>=`	True if `x` is greater than or equal to `y`.
`ToString(in x: Integer): String`	`ToString`	Converts `x` to a `String` value.
`ToUnlimitedNatural (in x: Integer): UnlimitedNatural[0..1]`	`ToUnlimited Natural`	Converts `x` to an `UnlimitedNatural` value.
`ToInteger(in x: String): Integer[0..1]`	`ToInteger`	Converts `x` to an `Integer` value.
`ToNatural(in x: String): Integer[0..1]`		Converts `x` to an `Integer` value, where `x` is any legal representation of a natural literal.
`ToReal(in x: Integer): Real`		Converts `x` to a `Real` value.

11.4.3 Real Functions

The `Alf::Library::PrimitiveBehaviors::RealFunctions` package imports all behaviors contained in the fUML `FoundationalModelLibrary::PrimitiveBehaviors::RealFunctions` package. Table 11.3 lists these functions with their Alf aliases, along with their corresponding fUML name. The formal specifications for these functions is given in 9.3.3 of the fUML Specification.

Table 11.3 Real Functions

Function Signature	fUML Name	Description
`Neg(in x: Real): Real`	`Neg`	The negative value of x.
`Abs(in x: Real): Real`	`Abs`	The absolute value of x.
`'+'(in x: Real, in y: Real): Real`	`+`	The value of the addition of x and y.
`'-'(in x: Real, in y: Real): Real`	`-`	The value of the subtraction of x and y.
`'*'(in x: Real, in y: Real): Real`	`*`	The value of the multiplication of x and y.
`'/'(in x: Real, in y: Real): Real[0..1]`	`/`	The value of the division of x and y.
`Max(in x: Real, in y: Real): Real`	`Max`	The maximum of x and y.
`Min(in x: Real, in y: Real): Real`	`Min`	The minimum of x and y.
`'<'(in x: Real, in y: Real): Boolean`	`<`	True if x is less than y.
`'>'(in x: Real, in y: Real): Boolean`	`>`	True if x is greater than y.
`'<='(in x: Real, in y: Real): Boolean`	`<=`	True if x is less than or equal to y.
`'>='(in x: Real, in y: Real): Boolean`	`>=`	True if x is greater than or equal to y.
`Floor(in x: Real): Integer[0..1]`	`Floor`	The largest integer that is less than or equal to x.
`Round(in x: Real): Integer[0..1]`	`Round`	The integer that is closest to x. (When there are two such integers, the largest one.)
`ToString(in x: Real): String`	`ToString`	Converts x to a `String` value.
`ToInteger(in x: Real): Integer[0..1]`	`ToInteger`	Converts x to an `Integer` value.
`ToReal(in x: String): Real[0..1]`	`ToReal`	Converts x to a `Real` value.

11.4.4 String Functions

The `Alf::Library::PrimitiveBehaviors::StringFunctions` package imports all the behaviors contained in the fUML `FoundationalModelLibrary::PrimitiveBehaviors:: StringFunctions` package. Table 11.4 lists these functions with their Alf aliases, along with their corresponding fUML name. The formal specification for these functions is given in 9.3.4 of the fUML Specification.

Table 11.4 `String` Functions

Function Signature	fUML Name	Description
`'+'(in x: String, in y: String): String`	`Concat`	The concatenation of `x` and `y`.
`Size(in x: String): Integer`	`Size`	The number of characters in `x`.
`Substring (in x: String, in lower: Integer, in upper: Integer): String[0..1]`	`Substring`	The substring of `x` starting at character number `lower`, up to and including character number `upper`. Character numbers run from 1 to `Size(x)`.

11.4.5 UnlimitedNatural Functions

The `Alf::Library::PrimitiveBehaviors::UnlimitedNaturalFunctions` package imports all the behaviors contained in the fUML `FoundationalModelLibrary::PrimitiveBehaviors::UnlimitedNaturalFunctions` package. Table 11.5 lists these functions with their Alf aliases, along with their corresponding fUML name. The formal specification for these functions is given in 9.3.5 of the fUML Specification.

Table 11.5 `UnlimitedNatural` Functions

Function Signature	fUML Name	Description
`Max (in x: UnlimitedNatural, in y: UnlimitedNatural): UnlimitedNatural`	`Max`	The maximum of `x` and `y`.
`Min (in x: UnlimitedNatural, in y: UnlimitedNatural): UnlimitedNatural`	`Min`	The minimum of `x` and `y`.
`'<' (in x: UnlimitedNatural, in y: UnlimitedNatural): Boolean`	`<`	True if `x` is less than `y`.
`'>' (in x: UnlimitedNatural, in y: UnlimitedNatural): Boolean`	`>`	True if `x` is greater than `y`.
`'<=' (in x: UnlimitedNatural, in y: UnlimitedNatural): Boolean`	`<=`	True if `x` is less than or equal to `y`.
`'>=' (in x: UnlimitedNatural, in y: UnlimitedNatural): Boolean`	`>=`	True if `x` is greater than or equal to `y`.
`ToString(in x: UnlimitedNatural): String`	`ToString`	Converts `x` to a `String` value.
`ToInteger (in x: UnlimitedNatural): Integer[0..1]`	`ToInteger`	Converts `x` to an `Integer` value.
`ToUnlimitedNatural(in x: String): UnlimitedNatural[0..1]`	`ToUnlimited Natural`	Converts `x` to an `UnlimitedNatural` value.

11.4.6 Bit String Functions

The `Alf::Library::PrimitiveBehaviors::BitStringFunctions` package contains a set of functions used to manipulate bit strings. Table 11.6 lists a basic set of functions included in the package for constructing and accessing bit strings. Table 11.7 lists functions corresponding to bit-wise operations for which there is a binary operator syntax.

Table 11.6 Basic `BitString` Functions

Function Signature	Description
`IsSet(in b: BitString, n: Integer): Boolean`	True if the `n`-th bit of `b` is set. Bits are numbered from *zero* starting with the *rightmost* bit position. If `n` is greater than or equal to the length of the bit string, then the result for the leftmost bit is returned.
`BitLength(): Integer`	The implemented bit string length.
`ToBitString(in n: Integer): BitString`	The bit string representation of `n`.
`ToInteger(in b: BitString): Integer`	The integer represented by the bit string `b`.
`ToHexString(in b: BitString): String`	The string representation of `b` as a hexadecimal numeral.
`ToOctalString(in b: BitString): String`	The string representation of `b` as an octal numeral.

There are no literals for bit strings, so the only way to construct a bit string is by using the function `toBitString`. This function takes an integer value and returns a bit string consisting of the twos-complement binary representation of that value, with the low-order bit being the rightmost bit (bit position 0). If an implementation supports arbitrary integer values, and the twos-complement representation of an integer value is longer than the bit string length for that implementation, then the result of `toBitString` consists of the low-order bits of the representation, truncated at the implemented bit string length.

Note that, in most cases, it is not necessary to call `toBitString` explicitly, because such a call is automatically inserted as a result of implicit bit string conversion of an integer value (see 8.8). The function toInteger performs the inverse conversion from a bit string to an integer value.

Alf provides special operator syntax for unary and binary bit-wise and shift functions on bit strings (sees 8.5.3, 8.6.7 and 8.6.3). This operator syntax is a shorthand for the invocation of the correspondingly named function behaviors given in Table 11.7. The required behavior of each of these functions is specified as a post-condition on its result, written in the Object Constraint Language (OCL), Version 2.2 (see the OCL Specification). The standard OCL notation is extended to allow calls to the functions `isSet` and `length` defined in Table 11.6.

Table 11.7 Bit-wise Operator Functions

Function Signature	Description

`'~'(in b: BitString): BitString`	The bit-wise complement of `b`.
	
	`Post:`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  IsSet(result,i)= not IsSet(b,i)`
	`  )`

`'&'(in b1: BitString, in b2: BitString):`
`  BitString`
	The bit-wise "and" of `b1` and `b2`.
	
	`Post:`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  isSet(result,i) =`
	`  isSet(b1,i) and isSet(b2,i)`
	`  )`

`'^'(in b1: BitString, in b2: BitString):`
`  BitString`
	The bit-wise "exclusive or" of `b1` and `b2`.
	
	`Post:`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  IsSet(result,i) =`
	`  IsSet(b1,i) xor IsSet(b2,i)`
	`  )`

`'|'(in b1: BitString, in b2: BitString):`
`  BitString`
	The bit-wise "inclusive or" of `b1` and `b2`.
	
	`Post:`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  IsSet(result,i) =`
	`  IsSet(b1,i) or IsSet(b2,i)`
	`  )`

`'<<'(in b: BitString, in n: Integer):`
`  BitString`
	The bit string b shifted n places to the left.
	
	`Post:`
	`  if n <= 0 then result = b`
	`  else`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  if i < n then`
	`  not IsSet(result,i)`
	`  else`
	`  IsSet(result,i) =`
	`  IsSet(b,i-n)`
	`  endif`
	`  )`
	`  endif`

Table 11.7 Bit-wise Operator Functions

Function Signature	Description

`'>>'(in b: BitString, in n: Integer): BitString`	The bit string `b` shifted `n` places to the right with "sign extension" of the leftmost bit.
	
	`Post:`
	`  if n <= 0 then result = b`
	`  else`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  if i < BitLength()-n then`
	`  IsSet(result,i) =`
	`  IsSet(b,i+n)`
	`  else`
	`  IsSet(result,i) =`
	`  IsSet(b,BitLength()-1)`
	`  endif`
	`  )`
	`  endif`

`'>>>'(in b: BitString, in n: Integer): BitString`	The bit string `b` shifted `n` places to the right.
	
	`Post:`
	`  if n <= 0 then result = b`
	`  else`
	`  Sequence{0..BitLength()-1}->`
	`  forAll(i |`
	`  if i < BitLength()-n then`
	`  IsSet(result,i) =`
	`  IsSet(b,i+n)`
	`  else`
	`  not IsSet(result,i)`
	`  endif`
	`  )`
	`  endif`

11.4.7 Sequence Functions

The `Alf::Library::PrimitiveBehaviors::SequenceFunctions` package contains a set of functions used to manipulate sequences of values. Table 11.8 lists function behaviors that are included in the `SequenceFunctions` package. The required behavior of each function is specified as a post-condition on its result, written in the Object Constraint Language (OCL), Version 2.2 (see the OCL Specification). In some cases, a pre-condition is also specified for a function. In this case, if the pre-condition is violated, then the function completes execution, but produces no output value. The result parameters for such functions are specified to have multiplicity 0..1 to allow for this.

Note that all the functions in Table 11.8 take a sequence as an `in` parameter and return some result based on that sequence. Since these functions are intended to apply to sequences of any type of value, the function parameters for such sequences and their elements are untyped. This, however, means that, using the normal typing rules, any type information on an input sequence to one of these functions is lost when the function is used, which can be inconvenient.

For example, suppose `integerList` is a sequence of type `Integer` and consider the application of the `Including` function to it to define an extended list. Unfortunately, the following is illegal:

```
let extendedList: Integer[] = Including(integerList, 1); // Type error!
```

Since the result of the call to `including` is untyped, it is not assignable to extendedList, which has type Integer. Instead, an explicit cast must be used:

```
let extendedList: Integer[] = (Integer)Including(integerList, 1); // Legal
```

To avoid having to do this, the template function behaviors in the `CollectionFunctions` package (see 11.6) should generally be used in preference to the primitive behaviors defined in the `SequenceFunctions` package. Note, however, that these template functions are actually defined in terms of the primitive sequence functions, since only non-template primitive behavior implementations can be registered with the fUML execution locus.

Table 11.8 Sequence Functions

Function Signature	Description

`Size`
`(in seq: any[*] sequence):`
`Integer`
	The number of elements in `seq`.
	
	`Post: result = seq->size()`

`Includes`
`(in seq: any[*] sequence,`
`in element: any):`
`Boolean`
	True if `element` is an element of `seq`, false otherwise.
	
	`Post: result = seq->includes(element)`

`Excludes`
`(in seq: any[*] sequence,`
`in element: any):`
`Boolean`
	True if `element` is not an element of `seq`, false otherwise.
	
	`Post: result = seq->excludes(element)`

`Count`
`(in seq: any[*] sequence,`
`in element: any):`
`Integer`
	The number of times that `element` occurs in `seq`.
	
	`Post: result = seq->count(element)`

`IsEmpty`
`(in seq: any[*] sequence):`
`Boolean`
	True if `seq` is empty.
	
	`Post: result = seq->isEmpty()`

`NotEmpty`
`(in seq: any[*] sequence):`
`Boolean`
	True if `seq` is not empty.
	
	`Post: result = seq->notEmpty()`

`IncludesAll`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence):`
`Boolean`
	True if `seq1` contains all the elements of `seq2`, false otherwise.
	
	`Post: result = seq1->includesAll(seq2)`

`ExcludesAll`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence):`
`Boolean`
	True if `c1` contains none of the elements of seq2, false otherwise.
	
	`Post: result = seq1->excludesAll(seq2)`

`Equals`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence):`
`Boolean`
	True if `seq1` contains the same elements as `seq2` in the same order.
	
	`Post: result = (seq1=seq2)`

`At`
`(in seq: any[*] sequence,`
`in index: Integer):`
`any[0..1]`
	The `index`-th element of `seq`.
	
	`Pre: index>=1 and index<=seq->size()`
	`Post: result = seq->at(index)`

Table 11.8 Sequence Functions

Function Signature	Description

`IndexOf`
`(in seq: any[*],`
`in element: any):`
`Integer[0..1]`
	The index of `element` in `seq`.
	
	`Pre: seq->includes(element)`
	`Post: seq->indexOf(element)`

`First`
`(in seq: any[*] sequence):`
`any[0..1]`
	The first element of `seq`.
	
	`Pre: c->notEmpty()`
	`Post: result = seq->first()`

`Last`
`(in seq: any[*] sequence):`
`any[0..1]`
	The last element of `seq`.
	
	`Pre: seq->notEmpty()`
	`Post: result = seq->last()`

`Union`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence):`
`any[*] sequence`
	The sequence consisting of all elements of `seq1` followed by all elements of `seq2`.
	
	`Post: result = seq1->union(seq2)`

`Intersection`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence):`
`any[*] sequence`
	The sequence consisting of all elements of `seq1` that are also in `seq2`.
	
	`Post: result = seq1->iterate(`
	`e; s = Sequence{} |`
	`if seq2->excludes(e) then s`
	`else s->append(e) endif)`

`Difference`
`(in seq1: any[*] sequence,`
`in seq2: any[*] sequence ):`
`any[*] sequence`
	The sequence consisting of all the elements of `seq1` not in `seq2`.
	
	`Post: result = seq1->iterate(`
	`e; s = Sequence{} |`
	`if seq2->includes(e) then s`
	`else s->append(e) endif)`

`Including`
`(in seq: any[*] sequence,`
`in element: any):`
`any[*]sequence`
	The sequence consisting of all elements of `seq` followed by `element`.
	
	`Post: result = seq->including(element)`

`IncludeAt`
`(in seq: any[*] sequence,`
`in index: Integer,`
`in element: any):`
`any[*] sequence`
	The sequence consisting of `seq` with `element` inserted at position `index`.
	
	`Post: result =`
	`if index>=1 and index<=seq->size()+1`
	`then`
	`seq->insertAt(index,element)`
	`else`
	`seq`
	`endif`

Table 11.8 Sequence Functions

Function Signature	Description

`InsertAt`
`(in seq: any[*] sequence,`
`in index: Integer,`
`in element: any):`
`any[*]sequence`
	The sequence consisting of `seq` with `element` inserted at position `index`. (This is the same as _includeAt. _insertAt is included for consistency with the similar OCL operation on sequences.)
	
	`Post: result =`
	`if index>=1 and index<=seq->size()+1`
	`then`
	`seq->insertAt(index,element)`
	`else`
	`seq`
	`endif`

`IncludeAllAt`
`(in seq1: any[*] sequence,`
`in index: Integer,`
`in seq2: any[*] sequence:`
`any[*] sequence`
	The sequence consisting of `seq1` with all elements of `seq2` inserted at position `index`.
	
	`Post: result =`
	`if index>=1 and index<=seq->size()+1`
	`then`
	`seq1->subSequence(1,index-1)->`
	`union(seq2)->`
	`union(seq1->subsequence(index,`
	`seq1->size()))`
	`else`
	`seq`
	`endif`

`Excluding`
`(in seq: any[*] sequence,`
`in element: any):`
`any[*] sequence`
	The sequence consisting of all elements of `seq` apart from all occurrences of `element`.
	
	`Post: result = seq->iterate(`
	`e; s = Sequence{} |`
	`if e=element then s`
	`else s->append(e) endif)`

`ExcludingOne`
`(in seq: any[*] sequence,`
`in element: any):`
`any[*] sequence`
	The sequence consisting of `seq` with the first occurrence of element (if any) removed.
	
	`Post: result =`
	`if seq->includes(element) then`
	`let index = seq->indexOf(element)`
	`in`
	`seq->subSequence(1,index-1)->`
	`union(seq->`
	`subSequence(index+1,seq->size()))`
	`else`
	`seq`
	`endif`

Table 11.8 Sequence Functions

Function Signature	Description

`ExcludeAt`
`(in seq: any[*] sequence,`
`in index: Integer):`
`any[*] sequence`
	The sequence consisting of `seq` with the `index`-th element removed.
	
	`Post: result =`
	`if index>=1 and index<=c->size then`
	`seq->subSequence(1,index-1)->`
	`union(seq->`
	`subSequence(index+1,seq->size()))`
	`else`
	`seq`
	`endif`

`Replacing`
`(in seq: any[*] sequence,`
`in element: any,`
`in newElement: any):`
`any[*] sequence`
	The sequence consisting of `seq` with all occurrences of `element` replaced with `newElement`.
	
	`Post: result->size() = seq->size() and`
	`Sequence{1..seq->size()}->forAll(i |`
	`result->at(i) =`
	`if seq->at(i) = element then`
	`newElement`
	`else`
	`seq->at(i)`
	`endif`

`ReplacingOne`
`(in seq: any[*] sequence,`
`in element: any,`
`in newElement: any):`
`any[*] sequence`
	The sequence consisting of `seq` with the first occurrence of `element` (if any) replaced with `newElement`.
	
	`Post: result =`
	`if seq->excludes(element) then seq`
	`else`
	`let index=seq->indexOf(element) in`
	`seq->subSequence(1,index-1)->`
	`union(seq->`
	`subSequence(index,seq->size()))`
	`endif`

`ReplacingAt`
`(in seq: any[*] sequence,`
`in index: Integer,`
`in element: any):`
`any[*] sequence`
	The sequence consisting of seq with the element at position `index` replaced with the given `element`.
	
	`Pre: index>=1 and index<=seq->size()`
	`Post: result =`
	`seq->subSequence(1,index-1)->`
	`union(seq->`
	`subSequence(index,seq->size()))`

Table 11.8 Sequence Functions

Function Signature	Description

`Subsequence`
`  (in seq: any[*] sequence,`
`  in lower: Integer,`
`  in upper: Integer):`
`  any[*] sequence`
	The sub-sequence of `seq` consisting of the elements at position `lower` up to and including position `upper`. If `upper` is larger than the size of `seq`, then it is treated as if it was equal to the size.
	
	`Pre: lower<=upper`
	`Post: result =`
	`  seq->subSequence(lower.max(1),`
	`  upper.min(seq->size()))`

`ToOrderedSet`
`  (in seq: any[*] sequence):`
`  any[*] ordered`
	The sequence `seq` with all duplicates removed.
	
	`Post:`
	`  result = seq->asOrderedSet()`

11.5 Basic Input and Output

As shown in Figure 11.1, the `Alf::Library::BasicInputOutput` package imports the fUML standard `FoundationalModelLibrary::BasicInputOutput` package (see fUML Specification, 9.5). This makes the fUML input-output classes available via the `Alf::Library::BasicInputOutput` namespace. In addition, the `FoundationalModelLibrary::BasicInputOutput` package imports the `FoundationalModelLibrary::Common package`, so the common classes from that package are also available via the `Alf::Library::BasicInputOutput` namespace.

11.6 Collection Functions

The `Alf::Library::CollectionFunctions` package contains template versions of the sequence functions defined in the `Alf::Library::PrimitiveBehaviors::SequenceFunctions` package. Each collection function has a single template parameter, the type of values in the sequence be operated on. The type inference rule for the invocation of template behaviors (see 8.3.9) then allows these behaviors to be invoked without having to explicitly notate the binding of their template parameter.

For example, if `integerList` is a sequence of type `Integer`, then the statement

```
let extendedList: Integer[] = including(integerList, 1);
```

is equivalent to

```
let extendedList: Integer[] = including<Integer>(integerList, 1);
```

Since the result type of `including<Integer>` is `Integer`, the right-hand side of this statement is assignable to the left-hand side without the need for an explicit cast.

Each of the functions defined in the `SequenceFunctions package` has a template version defined as an activity in the `CollectionFunctions` package. The body of this activity simply calls the corresponding primitive function and returns the result of that call, cast to the appropriate result type. For example, the `including` function may be defined as follows as an Alf unit (see 10.4.8 on activity definitions):

```
namespace Alf::Library::CollectionFunctions;
activity including<T>(in seq: T[*] sequence, in element: T):
  T[*] sequence
{
  return (T)PrimitiveBehaviors::SequenceFunctions::Including(seq, element);
```

```
}
```

Other functions are defined similarly.

NOTE. According to the copy semantics for templates (see 6.3), a call to a bound template activity, such as `including<Integer>`, has the semantics of a call to an effective bound element constructed from a copy of the template activity with each occurrence of the template parameter `T` replaced by the actual argument type. However, once static analysis is complete, a call to such a bound activity will have a semantically equivalent effect (in the sense defined in 2.3) to a direct call to the corresponding sequence function, without even a need for any cast (since the definitions of the sequence functions guarantee that their results have the expected dynamic type, even if the statically declared type is `any`). Therefore, a compliant compilative implementation may, if desired, produce a target fUML model that replaces calls to bound collection functions with direct calls to the corresponding sequence functions.

All the primitive sequence functions take a sequence as an `in` parameter and return some result based on that sequence (see 11.4.7) and, therefore, so do the corresponding collection functions. The collection functions may also take collection objects (see 11.7) as inputs because of the collection conversion rule for assignability (see 8.8). This is particularly useful when using a collection as the source in the sequence operation notation (see 8.3.17). The returned result, however, is always a sequence, though this may be used to construct a new collection object.

The `CollectionFunctions` package also includes a number of additional activities that take a sequence as an `inout` parameter and make changes to that sequence "in place". Table 11.9 lists these behaviors. The bodies of these in-place behaviors are defined to be an Alf assignment expression, generally involving one of the base set of collection functions. For example, the full definition of the `add` function, as an Alf unit, is

```
namespace Alf::Library::CollectionFunctions;
activity add<T>(inout seq: T[*] sequence, in element: T):
  T[*] sequence
{
  return seq = including(seq,element);
}
```

The in-place functions cannot be used with collection objects, since the inout parameter cannot be assigned back to the collection object. However, all collection classes have regular operations that correspond to the functionality of the "in place" functions (see 11.7). The in-place collection functions are intended to provide similar functionality for sequences to that provided by the similarly named operations on collection objects.

Table 11.9 Collection "In-Place" Behaviors

Activity Signature	Description

`add<T>`
`(inout seq: T[*] ordered,`
`in element: T):`
`T[*] sequence`
	Append `element` to the end of `seq`.
	
	`seq = including(seq,element)`

`addAll<T>`
`(inout seq1: T[*] sequence,`
`in seq2: T[*] sequence):`
`T[*] sequence`
	Append all elements of `seq2` to `seq1`.
	
	`seq1 = union(seq1,seq2)`

`addAt<T>`
`(inout seq: T[*] sequence,`
`in index: Integer,`
`in element: T):`
`T[*] sequence`
	Insert `element` at position `index` of `seq`.
	
	`seq = includeAt(seq,index,element)`

`addAllAt<T>`
`(inout seq1: T[*] sequence,`
`in index: Integer,`
`in seq2: T[*] sequence):`
`T[*] sequence`
	Insert all elements of `seq2` into `seq1` at position `index`.
	
	`seq = includeAllAt(seq,index,element)`

`remove<T>`
`(inout seq: T[*] sequence,`
`in element: T): T[*] sequence`
	Remove all occurrences of `element` from `seq`.
	
	`seq = excluding(seq,element)`

`removeAll<T>`
`(inout seq: T[*] sequence,`
`in in seq2 : T[*] sequence):`
`T[*] sequence`
	Remove all elements of `seq2` from `seq1`.
	
	`seq1 = difference(seq1,seq2)`

`removeOne<T>`
`(inout seq: T[*] sequence,`
`in element: T): T[*] sequence`
	Remove the first occurrence of `element` (if any) from `seq`.
	
	`seq = excludingOne(seq,element)`

`removeAt<T>`
`(inout seq: T[*] sequence,`
`in index: Integer):`
`T[*] sequence`
	Remove the element at position `index` from `seq`.
	
	`seq = excludeAt(seq,index)`

`replace<T>`
`(inout seq: T[*] sequence,`
`in element: T,`
`in newElement: T):`
`T[*] sequence`
	Replace all occurrences of `element` in `seq` with `newElement`.
	
	`seq = replacing(seq,element,newElement)`

`replaceOne<T>`
`(inout seq: T[*] sequence,`
`in element: T,`
`in newElement: T):`
`T[*] sequence`
	Replace the first occurrence of `element in seq` (if any) with `newElement`.
	
	`seq =`
	`replacingOne(seq,element,newElement)`

`replaceAt<T>`
`(inout seq: T[*] sequence,`
`in index: Integer,`
`in element: T):`
`T[*] sequence`
	Replace the element at position `index` in `seq` with the given `element`.
	
	`seq = replacingAt(seq,index,element)`

`clear<T>` `(inout seq: T[*])`	Clear all elements of `seq`. `seq = null`

11.7 Collection Classes

The `Alf::Library::CollectionClasses` package contains a set of template classes (see 6.3 on the semantics of templates) related to collections. These provide the ability to create and manipulate collections in a manner familiar from object-oriented programming languages. They may be used instead of or in conjunction with the basic use of UML sequences and functions to handle collections. Figure 11.5 is a class diagram of the contents of the `CollectionClasses` package.

In many places in Alf, collection objects may be used in the same way as a basic UML sequence of values. For example, the Alf assignability rules provide for *collection conversion* in which a collection object is automatically converted to a sequence via an implicit call to its `toSequence()` operation (see 8.8). In the main body of the specification, a collection object is any object whose type is a *collection class.* A collection class is any class that has a constructor of the same name with a single in parameter that is a sequence and an operation called `toSequence` that returns a sequence of the same type as the constructor input (known as the collection class's *sequence type*) and no other parameters. A template binding of any of the public template classes contained in the `CollectionClasses` package (other than the `Collection` superclass) conforms to this definition of collection class. However, it is allowable for a modeler to define other collection classes meeting the above definition, possibly, but not necessarily, as subclasses of template bindings of standard collection classes.

Note that all the classes shown in Figure 11.5 are abstract, but that they all also define constructor operations. Using the mechanism described in 8.3.12, the concrete implementations for these classes are defined in a packaged named `Impl` nested in the `CollectionClasses` package, as shown in Figure 11.6. With this mechanism, user models can reference and use the standard abstract classes as if they were concrete, and specific execution tools can define different implementations for these classes without affecting the classes actually referenced by user models.

A conformant implementation of the `CollectionClasses` package is not allowed to alter the definition of the classes shown in Figure 11.5. Rather, such an implementation must implement each of the `Impl` classes shown in Figure 11.6. This may be done either by providing external library implementations for the classes, through a mechanism specific to the execution tool, or by extending the model shown in Figure 11.6 by providing a fUML-conformant method for each of the public operations of the classes. In doing the latter, a conformant implementation is also allowed to:

- Replace public owned operations with inherited operations of the same signature.
- Add additional features and other members to the classes, as long as their visibility is `private` or `protected`, as well as additional private members to the `Impl` package.

B.4 gives a sample non-normative implementation of the `CollectionClasses::Impl` package as an Alf unit. The following subclauses describe each of the abstract template classes shown in Figure 11.5.

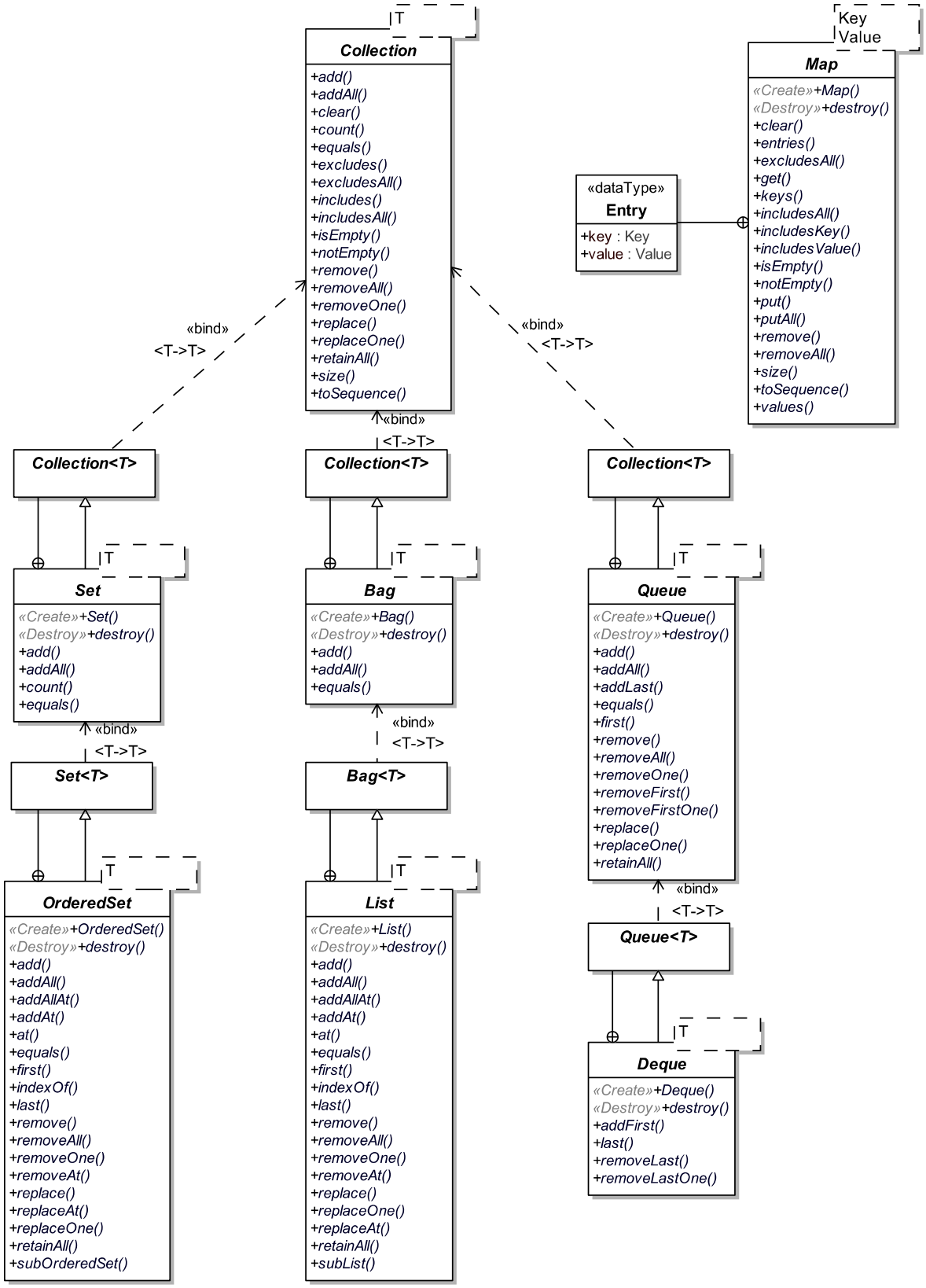

Figure 11.5 Collection Classes

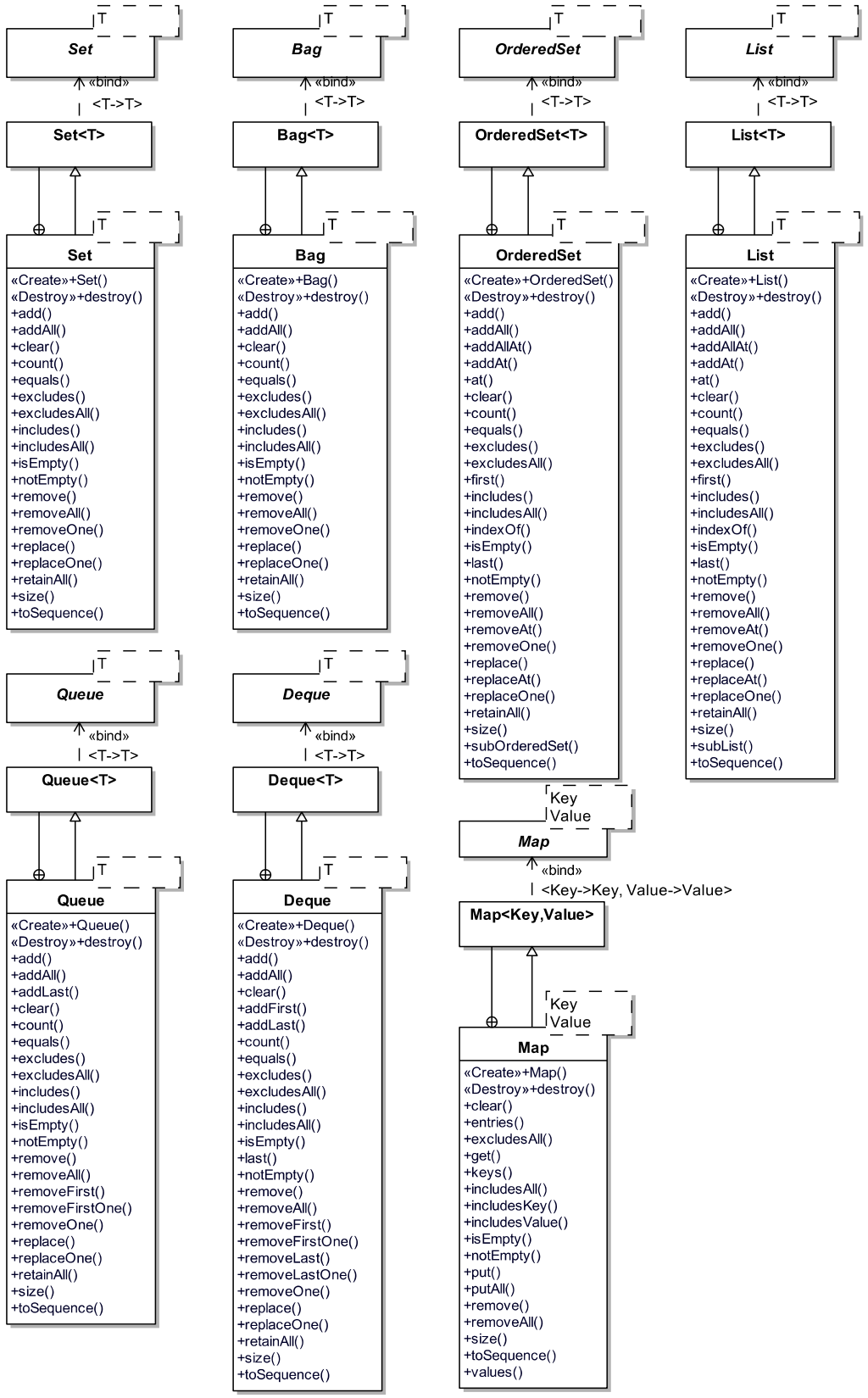

Figure 11.6 Collection Class Implementations

11.7.1 Bag<T>

Concrete unordered, non-unique collection. Supports duplicate entries.

Generalizations

- Collection<T>

Operations

[1] add (in element : T) : Boolean
Insert the given element into this bag. Always returns true.

post: self.toSequence()->asBag() = self@pre.toSequence()->asBag()->including(element)

[2] addAll (in seq : T [0..*] sequence) : Boolean
Insert all elements in the given sequence into this bag. Return true if the given sequence is not empty.

post: self.toSequence()->asBag() = self@pre.toSequence()->asBag()->union(seq->asBag())

[3] Bag (in seq : T [0..*] sequence) : Bag<T>
Construct a bag and add all elements in the given sequence.

post: result.toSequence()->asBag() = seq->asBag()

[4] destroy ( )
Destroy this bag.

[5] equals (in seq : T [0..*] sequence) : Boolean {query}
Return true if the content of this bag is equal to the given sequence considered as a bag.

post: result = (self@pre.toSequence()->asBag() = seq->asBag())

11.7.2 Collection<T>

An abstract collection of elements of a specified type <T>. Various concrete subclasses support ordered and unordered collections, with and without duplicates allowed.

Generalizations

None

Operations

[1] add (in element : T) : Boolean
Insert the given element into this collection. Return true if a new element is actually inserted.

post: result = self.size() > self@pre.size() and
result implies self.count(element) = self@pre.count(element)+1

[2] addAll (in seq : T [0..*] sequence) : Boolean
Insert all elements in the given sequence into this collection. Returns true if this collection increased in size.

post: result = self.size() > self@pre.size() and
self.includesAll(seq)

[3] clear ( )
Remove all elements from this collection.

post: result = self.isEmpty()

[4] count (in element : T) : Integer {query}
Return the number of elements in this collection that match a specified element.

post: result = self.toSequence()->count(element)

[5] equals (in seq : T [0..*] sequence) : Boolean {query}
Return true if the content of this collection is equal to the given sequence.

post: result implies self.includesAll(seq)

[6] excludes (in element : T) : Boolean {query}
Return true if this collection does not contain the given element.

post: result = self.toSequence()->excludes(element)

[7] excludesAll (in seq : T [0..*] sequence) : Boolean {query}
Return true if all elements in the given sequence are not in this collection.

post: result = self.toSequence()->excludesAll(seq)

[8] includes (in element : T) : Boolean {query}
Return true if this collection contains the given element.

post: result = self.toSequence()->includes(element)

[9] includesAll (in seq : T [0..*] sequence) : Boolean {query}
Return true if all elements in the given sequence are also in this collection.

post: result = self.toSequence()->includesAll(seq)

[10] isEmpty ( ) : Boolean {query}
Return true if this collection contains no elements.

post: result = self.toSequence()->isEmpty()

[11] notEmpty ( ) : Boolean {query}

Return true if this collection contains at least one element.

post: result = self.toSequence()->notEmpty()

[12] remove (in element : T) : Integer

Remove all occurrences of the given element from this collection and return the count of elements removed removed.

```
post: result = self@pre.count(element) and
      self.size() = self@pre.size() - result and
      self@pre.toSequence()->forAll(e | self.count(e) =
            if e = element then 0
            else self@pre.count(e) endif )
```

[13] removeAll (in seq : T [0..*] sequence) : Boolean

Remove all occurrences of all elements in the given sequence from this collection. Return true if the size of this collection changes.

```
post: result = self.size() < self@pre.size() and
      self.toSequence()->asSet() = self@pre.toSequence()->asSet() - seq->asSet() and
      self.toSequence()->forAll(e | self.count(e) = self@pre.count(e))
```

[14] removeOne (in element : T) : Boolean

Remove one occurrence of the given element from this collection and return true if an occurrence of element was removed. If the collection is ordered, the first element will be removed.

```
post: result = self@pre.includes(element) and
      self.size() = self@pre.size() - (if result then 1 else 0) endif and
      self@pre.toSequence()->forAll(e | self.count(e) =
            if result and e = element then self@pre.count(e)-1
            else self@pre.count(e) endif )
```

[15] replace (in element : T, in newElement : T) : Integer

Replace all occurrences of the given element with a new element and return the count of replaced elements.

```
post: result = if element<>newElement then self@pre.count(element) else 0 endif and
      self.size() = self@pre.size() and
      self.toSequence()->forAll(e | self.count(e) =
             if e = newElement then self@pre.count(e)+result
            else self@pre.count(e) endif )
```

[16] replaceOne (in element : T, in newElement : T) : Boolean

Replace one occurrence of the given element with newElement and return true if an element was replaced. If the collection is ordered, this will be the first occurrence.

```
post: result = (self@pre.includes(element) and element<>newElement) and
      self.size() = self@pre.size() and
      self.toSequence()->forAll(e | self.count(e) =
            if result and e = element then self@pre.count(e)-1
            else if result and e = newElement then self@pre.count(e)+1
            else self@pre.count(e) endif endif )
```

[17] retainAll (in seq : T [0..*] sequence) : Boolean

Remove all instances of all elements in this collection that are NOT in the given sequence. Return true if the size of this collection changes.

```
post: result = self.size() < self@pre.size() and
      self.toSequence()->asBag() = self@pre.toSequence()->asBag()->intersection(seq->asSet())
```

[18] size ( ) : Integer {query}

Return the number of elements contained in this collection.

```
post: result = self@pre.toSequence()->size()
```

[19] toSequence ( ) : T [0..*] sequence {query}

Return a sequence (UML ordered, non-unique collection) containing the elements of this collection. If the specific kind of collection orders its elements, then the returned sequence will have this order. Otherwise the order of the elements in the returned sequence is arbitrary. (The requirements on the returned sequence from this operation are specified implicitly by the required behavior of the mutating operations on the various Collection subclasses.)

11.7.3 Deque<T>

Double-Ended Queue (pronounced "deck"). Concrete ordered, nonunique collection. Supports duplicate entries. Ordered by position. Insertion and removal can occur at the front or the back of a deque. Can operate as FIFO (in at back, out at front). Can operate as Stack (in at front/back, out at front/back).

Generalizations

- Queue<T>

Operations

[1] addFirst (in element : T) : Boolean

Add element into this deque at the front. Always returns true.

```
post: result = true and
      self.toSequence() = self@pre.toSequence()->prepend(element)
```

[2] Deque (in seq : T [0..*] sequence) : Deque<T>
Construct a deque and add the elements in the given sequence.

post: self.toSequence() = seq

[3] destroy ( )
Destroy this deque.

[4] last ( ) : T [0..1] {query}
Return, but do not remove, the element at the back of the queue, if one exists.

pre: self.notEmpty()
post: result = self.toSequence()->last()

[5] removeLast ( ) : T [0..1]
Remove and return the element at the back of the deque if one exists.

pre: self.notEmpty()
post: result = self@pre.toSequence()->last() and
    self.toSequence() = self@pre.toSequence->subSequence(1,self@pre.size()-1)

[6] removeLastOne (in element : T) : T [0..1]
Remove and return the last occurrence of the given element in this deque. If this deque is empty or the element is not found in this queue, return nothing.

pre: self.includes(element)
post: result = element and
    let revSeq = self@pre.toSequence()->reverse() in
    let index = revSeq.indexOf(element) in
    self.toSequence() = revSeq->subSequence(1,index-1)->union(revSeq->subSequence(index+1,revSeq->size()))->reverse()

11.7.4 Entry

An association of value to key. Note that entries are data values that are always passed by copy. Changing an entry returned outside of a map will NOT effect the association within the map.

Generalizations

None

Attributes

- key : Key

  The key for this association, used for lookup

- value : Value

  An optional value for this association

11.7.5 List<T>

Concrete ordered, nonunique collection. Supports duplicate entries. Ordered by position in list.

Generalizations

- Bag<T>

Operations

[1] add (in element : T) : Boolean

Append the given element into this list at the end. Always returns true.

post: self.toSequence() = self@pre.toSequence()->append(element)

[2] addAll (in seq : T [0..*] sequence) : Boolean

Append all elements in the given sequence onto the end of this list. Return true if the given collection is not empty.

post: self.toSequence() = self@pre.toSequence()->union(seq)

[3] addAllAt (in index : Integer, in seq : T [0..*] sequence) : Boolean

Insert all elements in the given sequence into this list at the given position index. Return true if the given collection is not empty.

pre: index >= 1 and index <= self.size()+1
post: result = self.size() > self@pre.size() and
self.toSequence() = Sequence{1..seq->size()}->iterate(i; s = self@pre.toSequence() | s->insertAt(index+i-1, sequence->at(i))

[4] addAt (in index : Integer, in element : T) : Boolean

Insert an element into this list at the given position index. Always return true.

pre: index > 1 and index <= self.size()+1
post: result = true and
self.toSequence() = self@pre.toSequence()->insertAt(index,element)

[5] at (in index : Integer) : T [0..1] {query}

Return the element at the given position index or nothing if there is no element at the given position.

pre: index > 0 and index <= self.size()
post: result = self@pre.toSequence()->at(index)

[6] destroy ( )

Destroy this list

[7] equals (in seq : T [0..*] sequence) : Boolean {query}
Return true if the content of this list is equal to the given sequence.

post: result = (self@pre.toSequence() = seq)

[8] first ( ) : T [0..1] {query}
Returns the first element in this list, if one exists

pre: self.notEmpty()
post: result = self@pre.toSequence()->first()

[9] indexOf (in element : T) : Integer [0..1] {query}
Return the position of the first occurrence of the given element in this list or nothing if the element is not included in this collection.

pre: self.includes(element)
post: result = self@pre.toSequence() -> indexOf(element)

[10] last ( ) : T [0..1] {query}
Returns the last element in this list, if one exists

pre: self.notEmpty()
post: result = self@pre.toSequence()->first()

[11] List (in seq : T [0..*] sequence) : List<T>
Construct a list and add all elements in the given sequence

post: result.toSequence() = seq

[12] remove (in element : T) : Integer
Remove all occurrences of the given element from this list and return the count of elements removed.

post: self.toSequence() = self@pre.toSequence()->excluding(element)

[13] removeAll (in seq : T [0..*] sequence) : Boolean
Remove all elements in the given sequence from this list. Return true if the size of this list changes.

post: self.toSequence()
= seq->iterate(element; s = self@pre.toSequence() | s->excluding(element))

[14] removeAt (in index : Integer) : T [0..1]
Remove the element at the given position index and shift all trailing elements left by one position. Return the removed element, or nothing if the index is out of bounds.

pre: index > 0 and index <= self.size()

```
post: result = self@pre.at(index) and
        let preSeq = self@pre.toSequence() in
        self.toSequence() = preSeq->subSequence(1, index-1)->union(preSeq->subSequence(index+1,
self@pre.size()))
```

[15] removeOne (in element : T) : Boolean

Remove first occurrence of the given element from this list and return true if an occurrence of element was removed.

```
post: self.toSequence() =
        let preSeq = self@pre.toSequence() in
        if result then
                let index = self@pre.indexOf(element) in
                self.toSequence() = preSeq->subSequence(1, index-1)->
union(preSeq->subSequence(index+1, self@pre.size()))
        else preSeq endif
```

[16] replace (in element : T, in newElement : T) : Integer

Replace all occurrences of the given element with a new element and return the count of replaced elements.

```
post: Sequence{1..self.size()}->forAll(i | self.at(i) =
                if self@pre.at(i) = element then newElement
                else self@pre.at(i) endif)
```

[17] replaceAt (in index : Integer, in element : T) : T [0..1]

Replace the element at the given position index with the given new element. Return the replaced element, or nothing if the index is out of bounds

```
pre: index > 0 and index <= self.size()
post: result = self@pre.at(index) and
        let preSeq = self@pre.toSequence() in
        self.toSequence() = preSeq->subSequence(1, index-1)->append(newElement)->union(preSeq-
>subSequence(index+1, self@pre.size()))
```

[18] replaceOne (in element : T, in newElement : T) : Boolean

Replace one occurrence of the given element with
newElement and return true if an element was replaced.

```
post: Sequence{1..self.size()}->forAll(i | self.at(i) =
                if result and i = self@pre.indexOf(element) then newElement
                else self@pre.at(i) endif)
```

[19] retainAll (in seq : T [0..*] sequence) : Boolean

Remove all instances of all elements in this list that are NOT in the given collection. Return true if the size of this collection changes.

post: self.toSequence() = (self@pre.toSequence()->asSet() - seq->asSet())->iterate(element; a = self@pre.toSequence() | seq->excluding(element))

[20] subList (in fromIndex : Integer, in toIndex : Integer) : List<T> {query}

Return a new list containing all elements of this list from the lower position index up to and including the upper position index.

```
post: if lower < 1 or upper > self.size() then
          result.toSequence()->empty()
      else
         result.toSequence() = self.toSequence()->subSequence(lower,upper)
      endif
```

11.7.6 Map<Key, Value>

Dictionary of key and value pairs called "entries". Concrete unordered, unique (by key) collection.

Generalizations

None

Operations

[1] clear ( )

Remove all entries in this map.

post: self.isEmpty()

[2] destroy ( )

Destroy this map.

[3] entries ( ) : Set<Entry> {query}

Return a set of copies of the entries in this map.

post: result.equals(self.toSequence())

[4] excludesAll (in entries : Entry [0..*]) : Boolean {query}

Returns true if this map contains none of the given entries.

post: result = self.toSequence()->excludesAll(entries)

[5] get (in key : Key) : Value [0..1] {query}

Returns the value associated with the given key, or nothing if there is no entry in this map with its key equal to key.

pre: self.keys().toSequence()->includes(key)
post: result = self.toSequence()->select(e | e.key = key).value

[6] includesAll (in entries : Entry [0..*]) : Boolean {query}
Returns true if this map contains all of the given entries.

post: result = self.entries().includesAll(entries)

[7] includesKey (in key : Key) : Boolean {query}
Return true if this map contains an entry with its key equal to the given key.

post: result = self.keys().includes(key)

[8] includesValue (in value : Value [0..1]) : Boolean {query}
Return true if an entry in this map has its value equal to value.

post: result = self.toSequence()->exists(e | e.value = value)

[9] isEmpty ( ) : Boolean {query}
Return true if this map contains no entries.

post: result = self.toSequence()->isEmpty()

[10] keys ( ) : Set<Key> {query}
Return a set of copies of the keys in this map.

post: result.equals(self.toSequence().key)

[11] Map (in entries : Entry [0..*]) : Map<Key,Value>
Construct a map and add the given entries. No two entries may have the same key.

pre: entries->isUnique(key)
post: result.toSequence()->asSet() = sequence->asSet()

[12] notEmpty ( ) : Boolean {query}
Return true if this map contains at least one entry.

post: result = self.toSequence()->notEmpty()

[13] put (in key : Key, in value : Value [0..1]) : Value [0..1]
Associate a value with a key, creating a new entry if necessary. Return the previously associated value, or nothing if this is a new entry.

```
post: result = self@pre.get(key) and
      self.toSequence().key->asSet() = self@pre.toSequence().key->asSet()->including(key) and
      self.toSequence()->isUnique(key) and
      self.keys().toSequence()->forAll(k | self.get(k) =
         if e.key = key then value else self@pre.get(k))
```

[14] putAll (in entries : Entry [0..*])

Add all the given entries to this map. Any entry with a key already present in this map replaces the previous entry in this map. No two of the given entries may have the same key.

```
pre: entries->isUnique(key)
post: self.toSequence().key->asSet() = self@pre.toSequence().key->asSet()->union(entries->asSet())
and
      self.toSequence()->isUnique(key) and
      self.keys().toSequence()->forAll(k | self.get(k) =
         if entries.key->includes(k) then entries->select(key=k)
         else self@pre.get(k))
```

[15] remove (in key : Key) : Value [0..1]

Remove any association of a value to the given key. Return the value previously associated with the key, or nothing if there was no previous entry for the key

```
pre: self.includesKey(key)
post: result = self@pre.get(key) and
      self.toSequence()->isUnique(key) and
      self.toSequence()->asSet() = self@pre.toSequence()->reject(e | e.key = key)->asSet()
```

[16] removeAll (in keys : Key [0..*])

Remove all associations of a value to any of the given keys.

```
post: self.toSequence()->isUnique(key) and
      self.toSequence()->asSet() = self@pre.toSequence()->reject(e | keys->includes(e.key))->asSet()
```

[17] size ( ) : Integer {query}

Returns the number of entries in this map.

```
post: result = self.toSequence()->size()
```

[18] toSequence ( ) : Entry [0..*] sequence {query}

Return a sequence (UML ordered, non-unique collection) containing copies all entries in this map. The order is arbitrary. (The requirements on the returned sequence from this operation are specified implicitly by the required behavior of the mutating operations of the Map class.)

[19] values ( ) : Bag<Value> {query}

Return a bag of copies of the values in this map. (A bag is returned, since a single value may be associated with more than one entry in the map.)

post: result.equals(self.toSequence().value)

11.7.7 OrderedSet<T>

Concrete ordered, unique collection. Does not support duplicate entries. Ordered by position.

Generalizations

- Set<T>

Operations

[1] add (in element : T) : Boolean

Append the given element into this ordered set at the end. Return true if a new element is actually inserted.

post: self.toSequence()->asOrderedSet() = self@pre.toSequence()->asOrderedSet()->append(element)

[2] addAll (in seq : T [0..*] sequence) : Boolean

Append all elements in the given sequence onto the end of this ordered set. Returns true if this collection increased in size.

post: self.toSequence()->asOrderedSet() = self@pre.toSequence()->union(seq->asOrderedSet())

[3] addAllAt (in index : Integer, in seq : T [0..*] sequence) : Boolean

Insert all elements in the given sequence into this ordered set at the given position index. Returns true if the size of the ordered set increases (that is, if at least some of the inserted elements were not duplicates of elements already in the set).

pre: index >= 1 and index <= self.size()+1
post: result = self.size() > self@pre.size() and
self.toSequence()->asOrderedSet() = Sequence{1..seq->size()}->iterate(i; set = self@pre.toSequence()->asOrderedSet() | set->insertAt(index+i-1, seq->at(i))

[4] addAt (in index : Integer, in element : T) : Boolean

Insert an element into this ordered set at the given position index. Return true if the element was actually added to the set.

pre: index > 1 and index <= self.size()+1
post: result = (self.size() = self@pre.size() + 1) and
self.toSequence()->asOrderedSet() = self@pre.toSequence()->asOrderedSet()->insertAt(index,element)

[5] at (in index : Integer) : T [0..1] {query}
Return the element at the given position index or nothing if there is no element at the given position.

pre: index > 0 and index <= self.size()
post: result = self@pre.toSequence()->at(index)

[6] destroy ( )
Destroy this ordered set.

[7] equals (in seq : T [0..*] sequence) : Boolean {query}
Return true if the content of this ordered set is equal to the given sequence considered as an ordered set.

post: result = (self@pre.toSequence()->asOrderedSet() = seq->asOrderedSet())

[8] first ( ) : T [0..1] {query}
Returns the first element in this ordered set, if one exists

pre: self.notEmpty()
post: result = self@pre.toSequence()->first()

[9] indexOf (in element : T) : Integer [0..1] {query}
Return the position of the first occurrence of the given element in this ordered set or nothing if the element is not included in this collection.

pre: self.includes(element)
post: result = self@pre.toSequence() -> indexOf(element)

[10] last ( ) : T [0..1] {query}
Returns the last element in this ordered set, if one exists

pre: self.notEmpty()
post: result = self@pre.toSequence()->last()

[11] OrderedSet (in seq : T [0..*] sequence) : OrderedSet<T>
Constructs an ordered set and adds all elements in the given sequence, in order.

post: result.toSequence()->asOrderedSet() = seq->asOrderedSet()

[12] remove (in element : T) : Integer
Remove all occurrences of the given element from this ordered set and return the count of elements removed. (For an ordered set, this has the same effect as removeOne, since duplicates are not allowed.)

post: self.toSequence()->asOrderedSet() = self@pre.toSequence()->asOrderedSet()->excluding(element)

[13] removeAll (in seq : T [0..*] sequence) : Boolean

Remove all elements in the given sequence from this ordered set. Return true if the size of this ordered set changes.

```
post: self.toSequence() = seq->iterate(element; s = self@pre.toSequence() | s->excluding(element))
```

[14] removeAt (in index : Integer) : T [0..1]

Remove the element at the given position index and shift all trailing elements left by one position. Return the removed element, or nothing if the index is out of bounds.

```
pre: index > 0 and index <= self.size()
post: result = self@pre.at(index) and
        self.toSequence() = self@pre.toSequence()->excluding(result)
```

[15] removeOne (in element : T) : Boolean

Remove one occurrence of the given element from this ordered set and return true if an occurrence of element was removed. (For an ordered set, this has the same effect as remove, since duplicates are not allowed.)

```
post: self.toSequence()->asOrderedSet() = self@pre.toSequence()->asOrderedSet()->excluding(element)
```

[16] replace (in element : T, in newElement : T) : Integer

Replace all occurrences of the given element with newElement and return the count of replaced elements. (For an ordered set, this has the same effect as replaceOne, since duplicates are not allowed.)

```
post: self.toSequence() = if result then
            self@pre.toSequence()->excluding(element)->insertAt(newElement,
self@pre.indexOf(element))
        else
            self@pre.toSequence()
        endif
```

[17] replaceAt (in index : Integer, in newElement : T) : T [0..1]

Replace the element at the given position index with the given new element. Return the replaced element, or nothing is the index is out of bounds

```
pre: index > 0 and index <= self.size()
post: result = self@pre.at(index) and
        self.toSequence() = self@pre.toSequence()->excluding(result)->insertAt(index,newElement)
```

[18] replaceOne (in element : T, in newElement : T) : Boolean

Replace one occurrence of the given element with newElement and return true if an element was replaced. (For an ordered set, this has the same effect as replace, since duplicates are not allowed.)

```
post: self.toSequence() = if result then
            self@pre.toSequence()->excluding(element)->insertAt(newElement,
self@pre.indexOf(element))
       else
            self@pre.toSequence()
      endif
```

[19] retainAll (in seq : T [0..*] sequence) : Boolean

Remove all instances of all elements in this ordered set that are NOT in the given sequence. Return true if the size of this collection changes.

```
post: self.toSequence() = (self@pre.toSequence()->asSet() - seq->asSet())->iterate(element; s =
self@pre.toSequence() | s->excluding(element))
```

[20] subOrderedSet (in lower : Integer, in upper : Integer) : OrderedSet<T> {query}

Return a new ordered set containing all elements of this ordered set from the lower position index up to and including the upper position index.

```
post: if lower < 1 or upper > self.size() then
          result.toSequence()->empty()
       else
         result.toSequence() = self.toSequence()->subSequence(lower,upper)
       endif
```

11.7.8 Queue<T>

First In First Out Queue. Concrete ordered, nonunique collection. Supports duplicate entries. Ordered by position. Considering the queue as a sequence, insertion occurs at the back of the queue, removal at the front.

Generalizations

- Collection<T>

Operations

[1] add (in element : T) : Boolean

Add the given element into this queue at the back. Always returns true.

```
post: self.toSequence() = self@pre.toSequence()->append(element)
```

[2] addAll (in seq : T [0..*] sequence) : Boolean

Add all elements in the given sequence to this queue at the back. Return true if the given collection is not empty.

```
post: self.toSequence() = self@pre.toSequence()->union(seq)
```

[3] addLast (in element : T) : Boolean

Add the given element into this queue at the back. Always returns true. (This is the same functionality as the add operation.)

```
post: result = true and
      self.toSequence() = self@pre.toSequence()->append(element)
```

[4] destroy ( )

Destroys this queue.

[5] equals (in seq : T [0..*] sequence) : Boolean {query}

Return true if the content of this queue is equal to the given sequence.

```
post: result = self@pre.toSequence() = seq
```

[6] first ( ) : T [0..1] {query}

Return, but do not remove, the element at the front of the queue, if one exists.

```
pre: self.notEmpty()
post: result = self.toSequence()->first()
```

[7] Queue (in seq : T [0..*] sequence) : Queue<T>

Construct a queue and add all elements in the given sequence.

```
post: result.toSequence() = seq
```

[8] remove (in element : T) : Integer

Remove all occurrences of the given element from this queue and return the count of elements removed.

```
post: self.toSequence() = self@pre.toSequence()->excluding(element)
```

[9] removeAll (in seq : T [0..*] sequence) : Boolean

Remove all elements in the given collection from this queue. Return true if the size of this queue changes.

```
post: self.toSequence()
 = seq->iterate(element; s = self@pre.toSequence() | s->excluding(element))
```

[10] removeFirst ( ) : T [0..1]

Remove and return the element at the front of the queue if one exists.

```
pre: self.notEmpty()
post: result = self@pre.toSequence()->first() and
      self.toSequence()->self@pre.toSequence()->subSequence(2,self@pre.size())
```

[11] removeFirstOne (in element : T [1]) : T [0..1]

Remove and return the first occurrence of the given element in this queue. If this queue is empty or the element is not found in this queue, return nothing.

```
pre: self.includes(element)
post: result = element and
        let preSeq = self@pre.toSequence() in
        let index = preSeq.indexOf(element) in
        self.toSequence() = preSeq->subSequence(1,index-1)->union(preSeq-
>subSequence(index+1,preSeq->size()))
```

[12] removeOne (in element : T) : Boolean

Remove the first occurrence of the given element from this queue and return true if an occurrence of element was removed.

```
post: self.toSequence() =
        let preSeq = self@pre.toSequence() in
        if result then
                let index = self@pre.indexOf(element) in
                self.toSequence() = preSeq->subSequence(1, index-1)->
union(preSeq->subSequence(index+1, self@pre.size()))
        else preSeq endif
```

[13] replace (in element : T, in newElement : T) : Integer

Replace all occurrences of the given element with a new element and return the count of replaced elements.

```
post: Sequence{1..self.size()}->forAll(i | self.at(i) =
                if self@pre.at(i) = element then newElement
                else self@pre.at(i) endif)
```

[14] replaceOne (in element : T, in newElement : T) : Boolean

Replace one occurrence of the given element with
newElement and return true if an element was replaced.

```
post: Sequence{1..self.size()}->forAll(i | self.at(i) =
                if result and i = self@pre.indexOf(element) then newElement
                else self@pre.at(i) endif)
```

[15] retainAll (in seq : T) : Boolean

Remove all instances of all elements in this queue that are NOT in the given collection. Return true if the size of this collection changes.

```
post: self.toSequence() = (self@pre.toSequence()->asSet() - seq->asSet())->iterate(element; s =
self@pre.toSequence() | s->excluding(element))
```

11.7.9 Set<T>

A concrete unordered, unique collection. Does not support duplicate entries.

Generalizations

- Collection<T>

Operations

[1] add (in element : T) : Boolean
Insert the given element into this set. Return true if a new element is actually inserted.

post: self.toSequence()->asSet() = self@pre.toSequence()->asSet()->including(element)

[2] addAll (in seq : T [0..*] sequence) : Boolean
Insert all elements in the given sequence into this set. Returns true if this collection increased in size.

post: self.toSequence()->asSet() = self@pre.toSequence()->asSet()->union(seq->asSet())

[3] count (in element : T) : Integer {query}
The number of elements in this set that match a specified element.

post: result = if self@pre.includes(element) then 1 else 0 endif

[4] destroy ( )
Destroy this set.

[5] equals (in seq : T [0..*] sequence) : Boolean {query}
Return true if the content of this set is equal to the given sequence considered as a set.

post: result = (self@pre.toSequence()->asSet() = seq->asSet())

[6] Set (in seq : T [0..*] sequence) : Set<T>
Construct a set and add all elements in the given sequence.

post: result.toSequence()->asSet() = seq->asSet()

12 Common Abstract Syntax

12.1 Overview

Parsing an Alf text synthesizes an abstract syntax tree (see 6.5). Static semantic analysis then adds associated with the syntax elements in the abstract syntax tree (see 6.6). Such information is formally specified using *derived* attributes and associations in the UML abstract syntax model. Thus, a syntax element class will generally have two kinds of properties: *synthesized properties,* whose values are determined during parsing, and *derived properties,* whose values are determined during static semantic analysis. The specification for each derived property includes a constraint that defines its derivation.

In addition, the specification of a syntax class includes various constraints that must be checked during the course of static semantics analysis. If any of these constraints are violated, then the input text is not legal and must be rejected. The specification of the class may also include helper operations that can be used in both the class constraints and property derivations.

Clauses 8, 9 and 10 described the synthesized abstract syntax for Alf expressions, statements and units. This clause extends that abstract syntax model to include the derived properties and constraints that specify the static semantics of Alf. Clauses 16 through 19 then specify the mapping from the Alf abstract syntax, after static semantic analysis, to the fUML subset of the UML abstract syntax.

The UML model of the Alf abstract syntax is contained within the package `Alf::Syntax`. It comprises four subpackages: `Common`, `Expressions`, `Statements` and `Units`. The content of each of these packages is described in this and the following three clauses.

The `Alf::Syntax::Common` package contains the root abstract classes `SyntaxElement` and `DocumentedElement`. It also contains common `ElementReference` and `AssignedSource` classes that are used throughout the rest of the abstract syntax model. Element reference and assigned source objects are not themselves syntax elements but, rather, represent certain relevant information determined during the static semantic analysis of an abstract syntax tree and associated with elements of that tree.

The mapping of common elements to UML is given in Clause 16.

Element References

As discussed in subclauses 8.2 and 10.2, a name resolves to the model element that it names. If this model element is defined outside the context of the Alf text in which the reference to its name appears, then the resolution is straightforward. However, the situation is more complicated if the element being referenced is defined within the same Alf text as the reference itself. This is because the name must be resolved during static semantic analysis, but, at that point, the Alf representation of the element being referenced has not yet been mapped to UML.

In order to handle this in a general way, the Alf abstract syntax uses a generic concept of *element reference.* Such a reference may be an *external* element reference, meaning that it is a reference to a model element defined in the UML model external to the Alf text. Or it may be an *internal* element reference, meaning that it is a reference to the syntax element in the Alf text which will ultimately be mapped to the desired model element. The necessary properties of a referenced element required for static semantic analysis may be obtained from either an external or internal element reference.

Assigned Sources

As discussed in subclauses 8.1 and 9.1, the assigned source for a local name is the statically determined syntax element that, when executed, will provide the actual assigned value for that local name. If the assigned source for a local name is known, then a reference to the assigned value of that local name can be mapped to an object flow from the mapping of the assigned source.

During static semantic analysis, it is necessary to know what names have assigned sources before and after each expression node and each statement node within the abstract syntax tree. Information on the assigned source includes not only the source element itself, but also the soriginally declared type, the statically best known subtype and the multiplicity of the value produced by that source.

A parameter name for an `in` or `inout` parameter may also have that parameter as its assigned source (see 8.3.3). If a parameter name is used in an Alf expression that is within a non-Alf unit, then the named parameter may be an external UML model element. As shown in Figure 12.3, the source of an assigned source is an element reference (rather than just an Alf syntax element), in order to allow for this case.

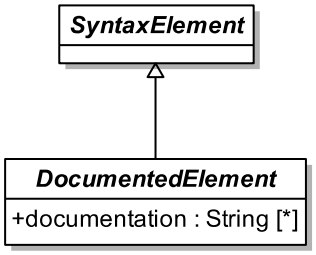

Figure 12.1 Syntax Elements

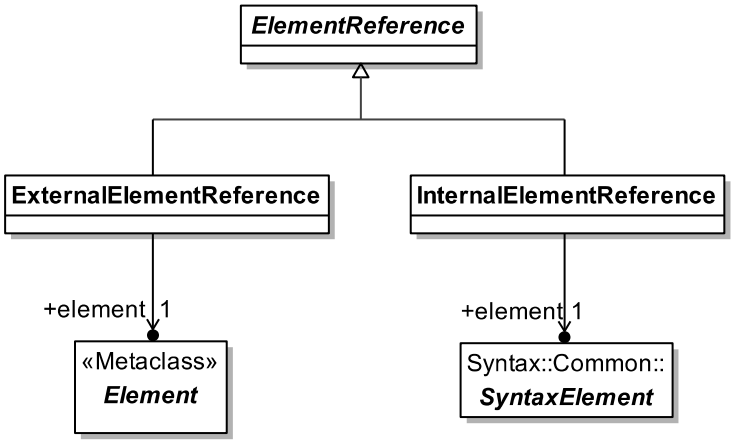

Figure 12.2 Element References

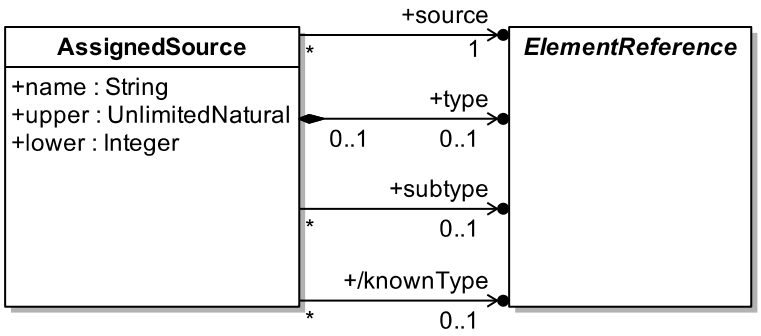

Figure 12.3 Assigned Sources

12.2 Class Descriptions

12.2.1 AssignedSource

An assignment of a source element that gives the value of a local name or input parameter name, along with a record of the defined type (if any) and multiplicity of the name.

Generalizations

None

Synthesized Properties

- lower : Integer

  The multiplicity lower bound for the name.

- name : String

  The local name for which this is the assigned source.

- source : ElementReference

  A reference to the element that is to be the source for the assigned value of the given local name.

- subtype : ElementReference [0..1]

  A reference to the element that gives the best known type for the latest assignment to the local name (if any). This will always be a subtype of the type of the assigned source.
- type : ElementReference [0..1]

  A reference to the element that gives the type for the local name (if any), as originally declared or assigned.

- upper : UnlimitedNatural

  The multiplicity upper bound for the local name.

Derived Properties

- knownType : ElementReference [0..1]

  A reference to the element that represents the best known type for the local name.

Constraints

[1] assignedSourceKnownTypeDerivation

If the subtype of an assigned source is empty, then its known type is the same as its type. Otherwise, its known type is the same as its subtype.

Helper Operations

None

12.2.2 DocumentedElement

A syntax element that has documentation comments associated with it.

Generalizations

- SyntaxElement

Synthesized Properties

- documentation : String [*]

  The documentation text to be associated with a syntax element. Each string is intended to be mapped to the body of a comment element in the target UML model.

Derived Properties

None

Constraints

None

Helper Operations

None

12.2.3 ElementReference

A reference to a model element, either directly or via its Alf abstract syntax representation. (NOTE: The definitions of all the helper operations of ElementReference are specific to its subclasses.)

Generalizations

None

Synthesized Properties

None

Derived Properties

None

Constraints

None

Helper Operations

None

12.2.4 ExternalElementReference

A direct reference to a UML model element.

Generalizations

- ElementReference

Synthesized Properties

- element : Element

  The referenced model element.

Derived Properties

None

Constraints

None

Helper Operations

None

12.2.5 InternalElementReference

A direct reference to a UML model element.

Generalizations

- ElementReference

Synthesized Properties

- element : SyntaxElement

  The Alf syntax element that represents the referenced model element.

Derived Properties

None

Constraints

None

Helper Operations

None

12.2.6 SyntaxElement

A syntax element synthesized in an abstract syntax tree, along with any additional information determined during static semantic analysis.

Generalizations

None

Synthesized Properties

None

Derived Properties
None

Constraints
None

Helper Operations
None

13 Expressions Abstract Syntax

13.1 Overview

The `Alf::Syntax::Expressions` package contains the abstract syntax model for expressions. The syntax and semantics of expressions are discussed in Clause 8. Their mapping to UML is given in Clause 17.

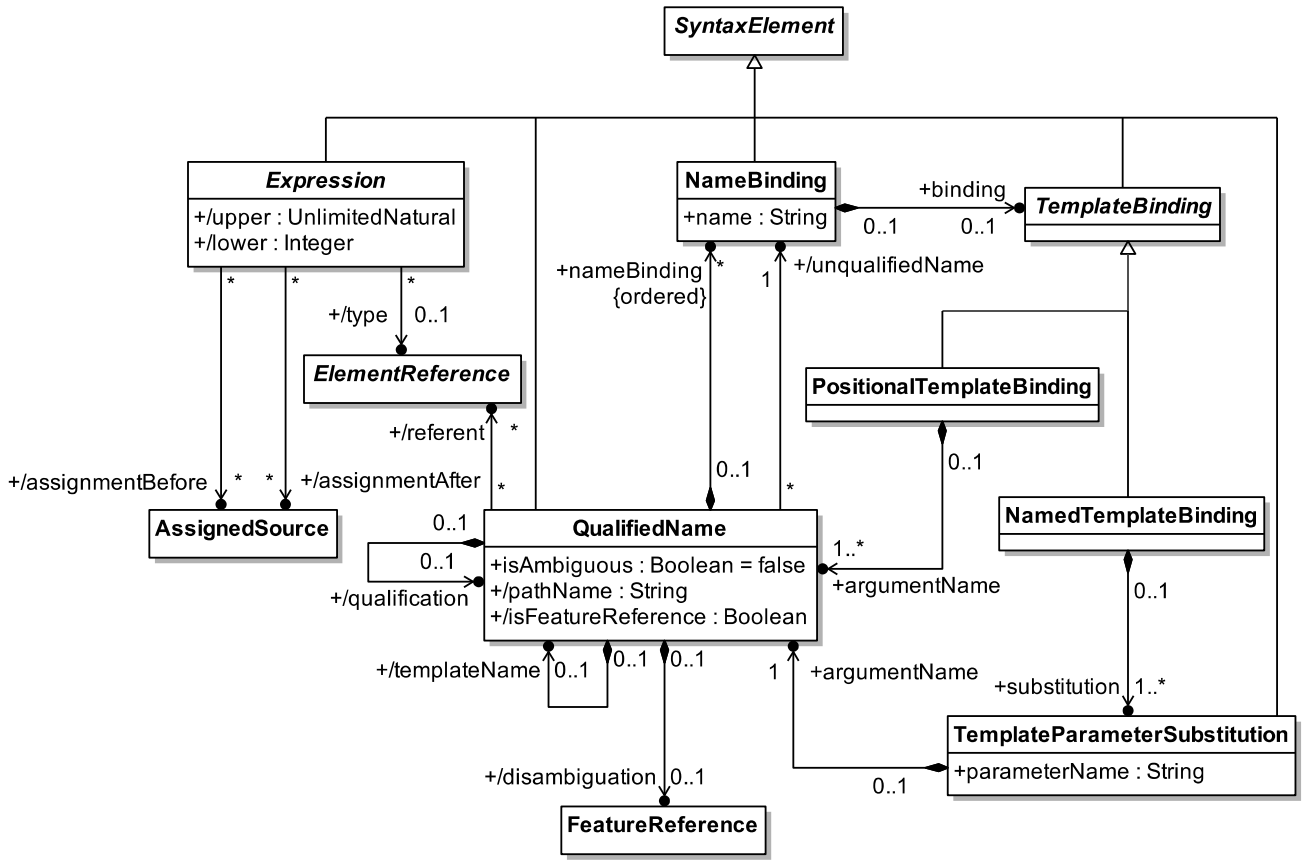

Figure 13.1 Expressions and Qualified Names

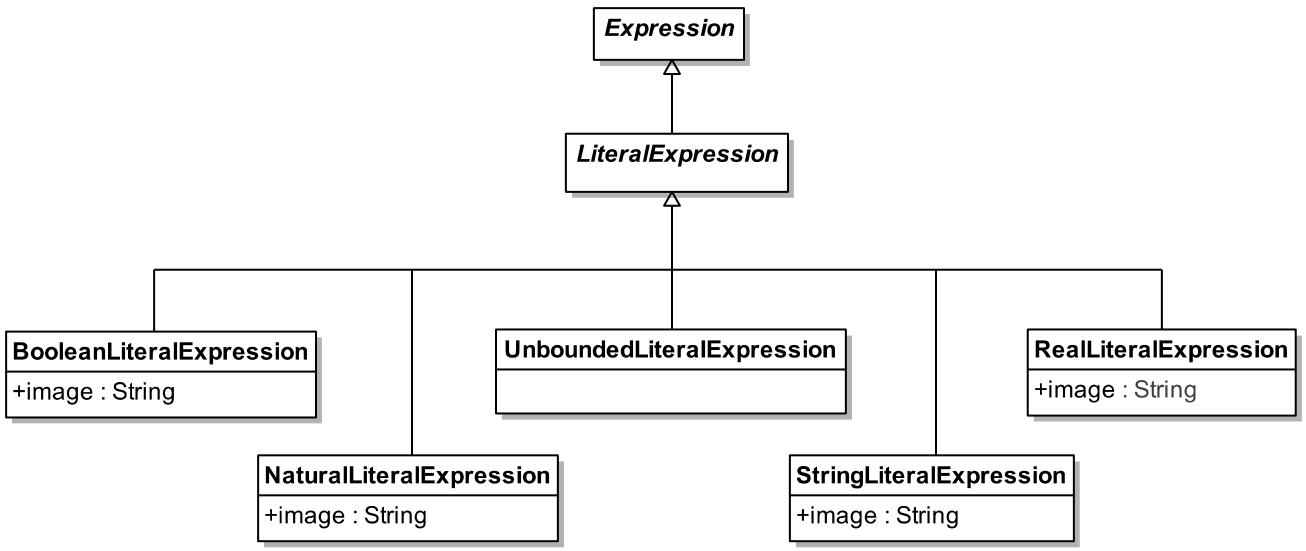

Figure 13.2 Literal Expressions

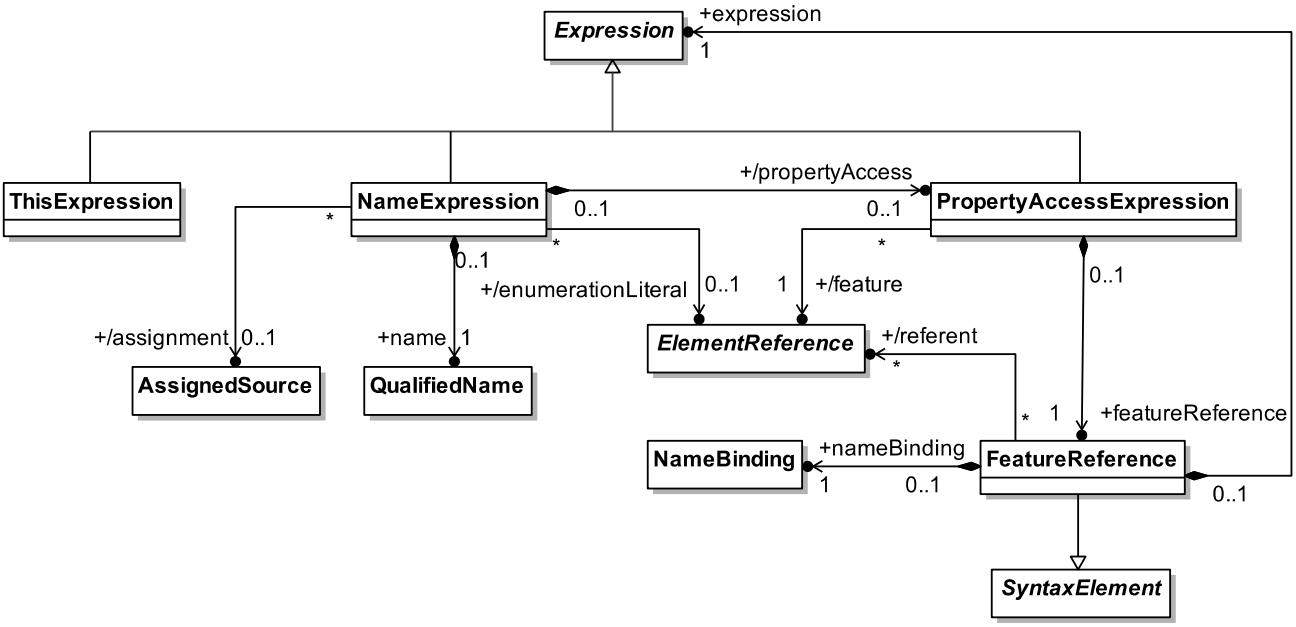

Figure 13.3 Basic Primary Expressions

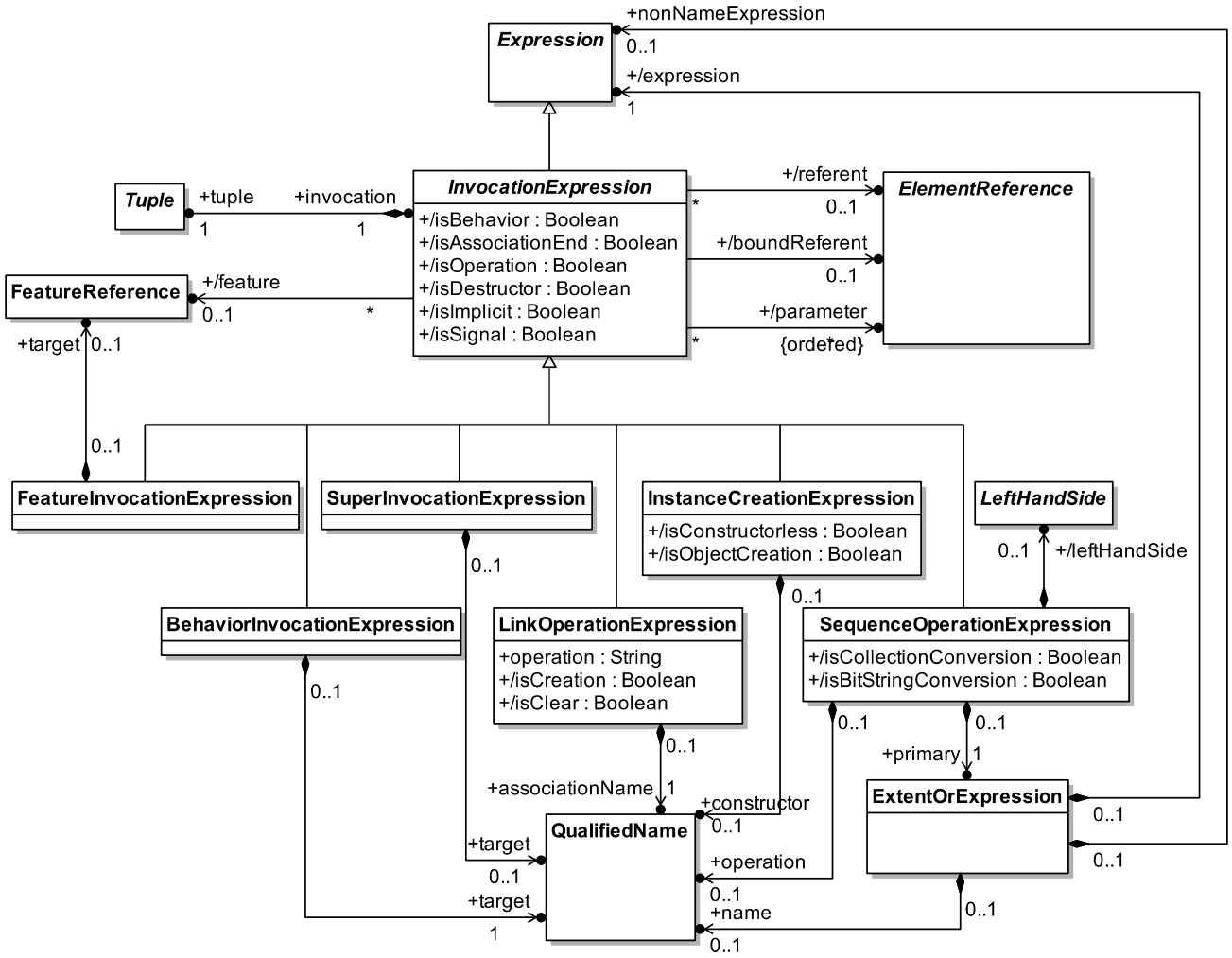

Figure 13.4 Invocation Expressions

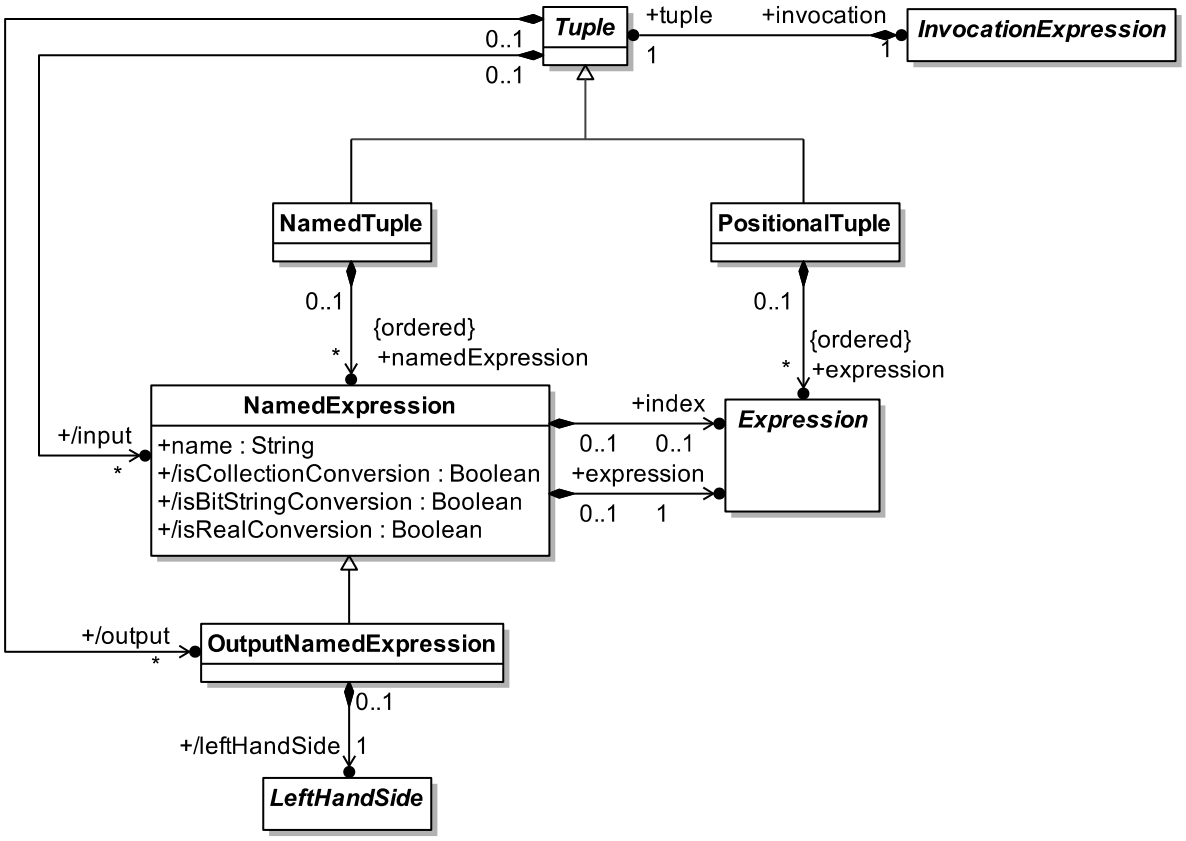

Figure 13.5 Tuples

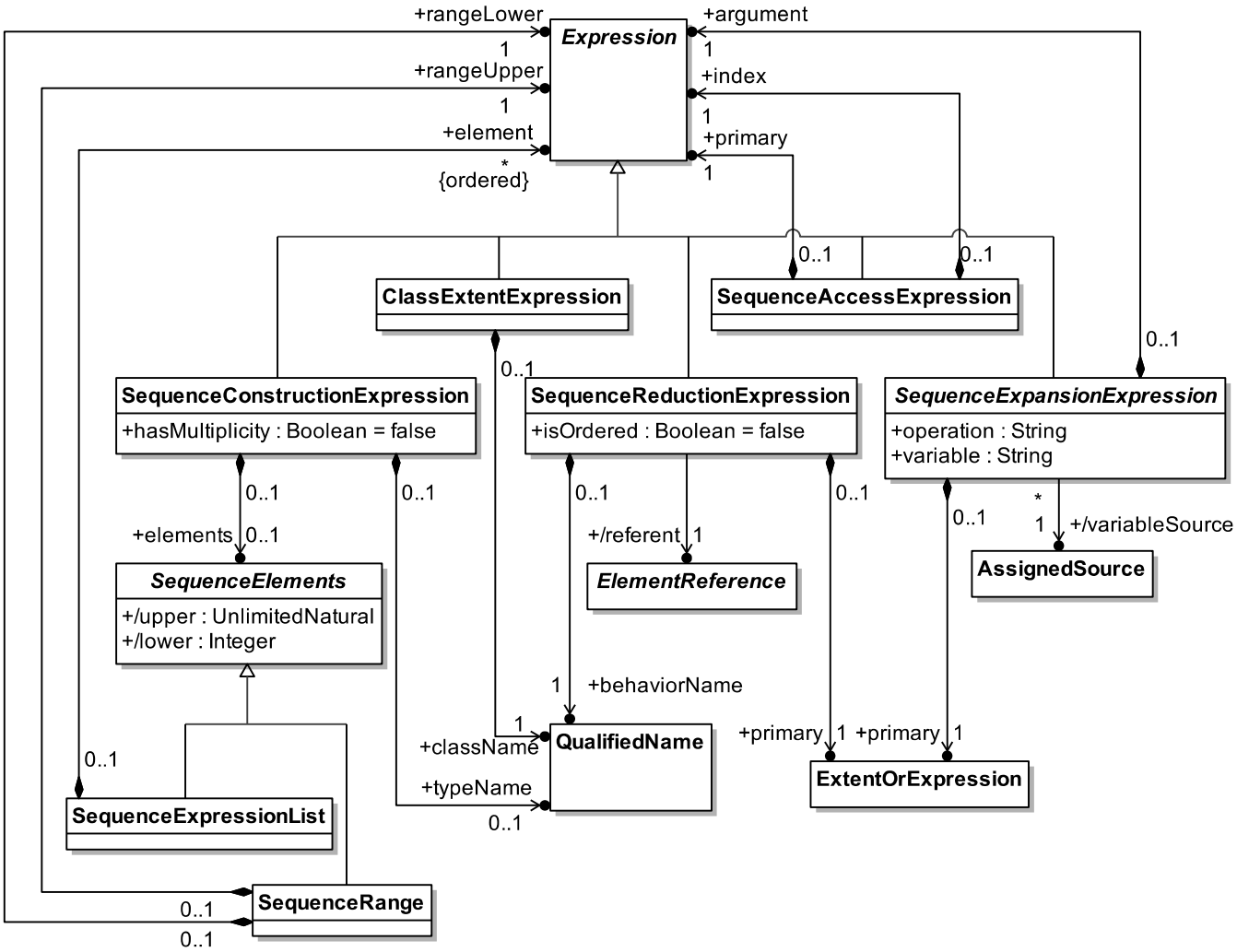

Figure 13.6 Sequence Expressions

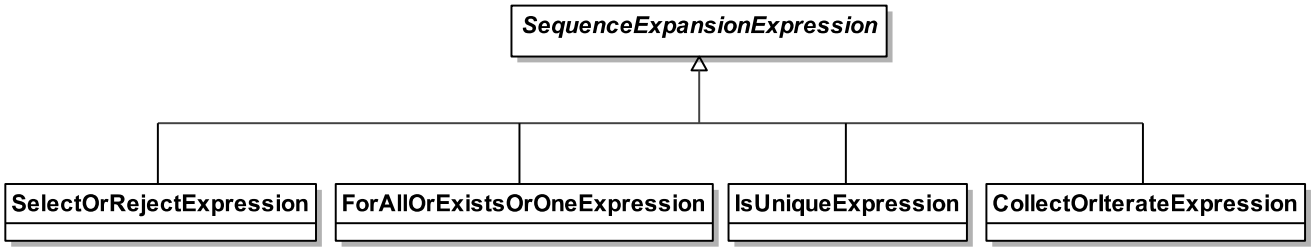

Figure 13.7 Sequence Expansion Expressions

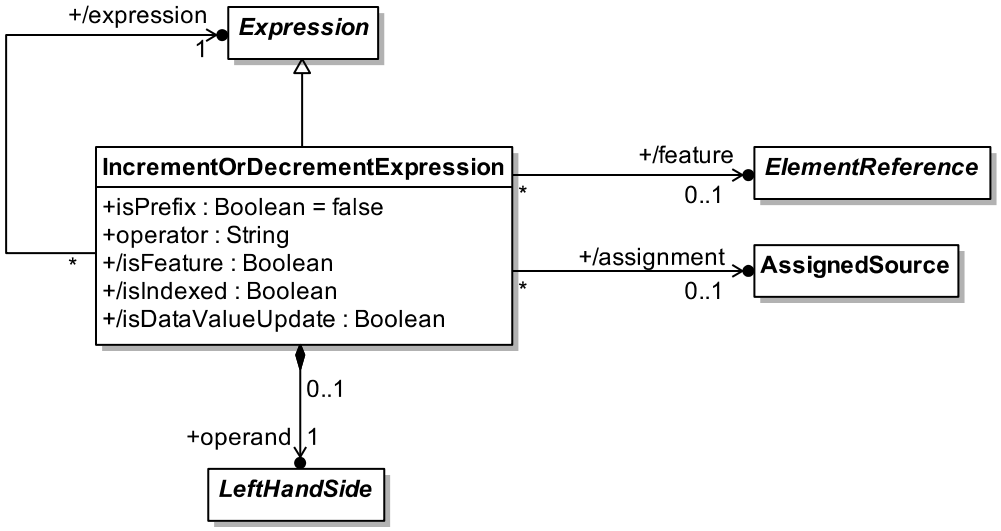

Figure 13.8 Increment and Decrement Expressions

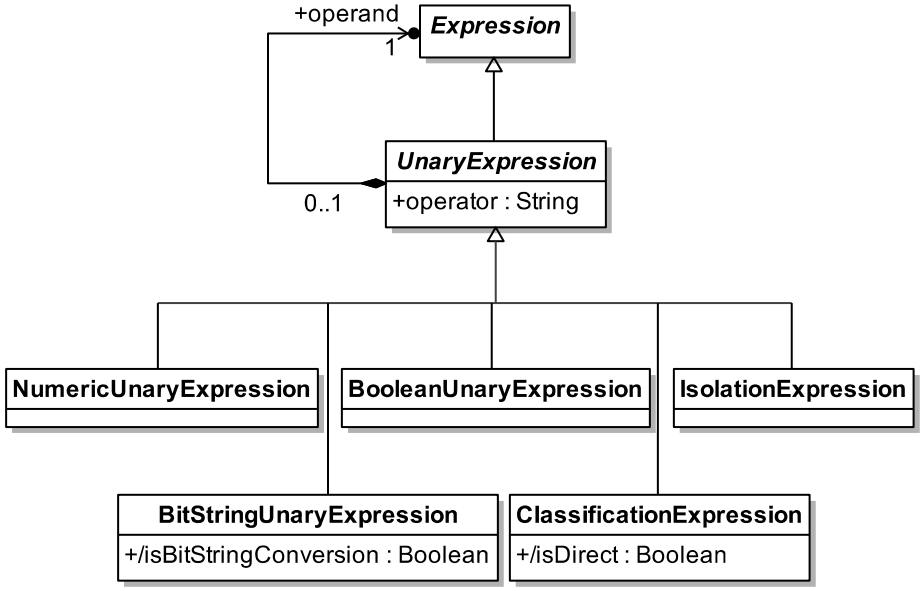

Figure 13.9 Unary Expressions

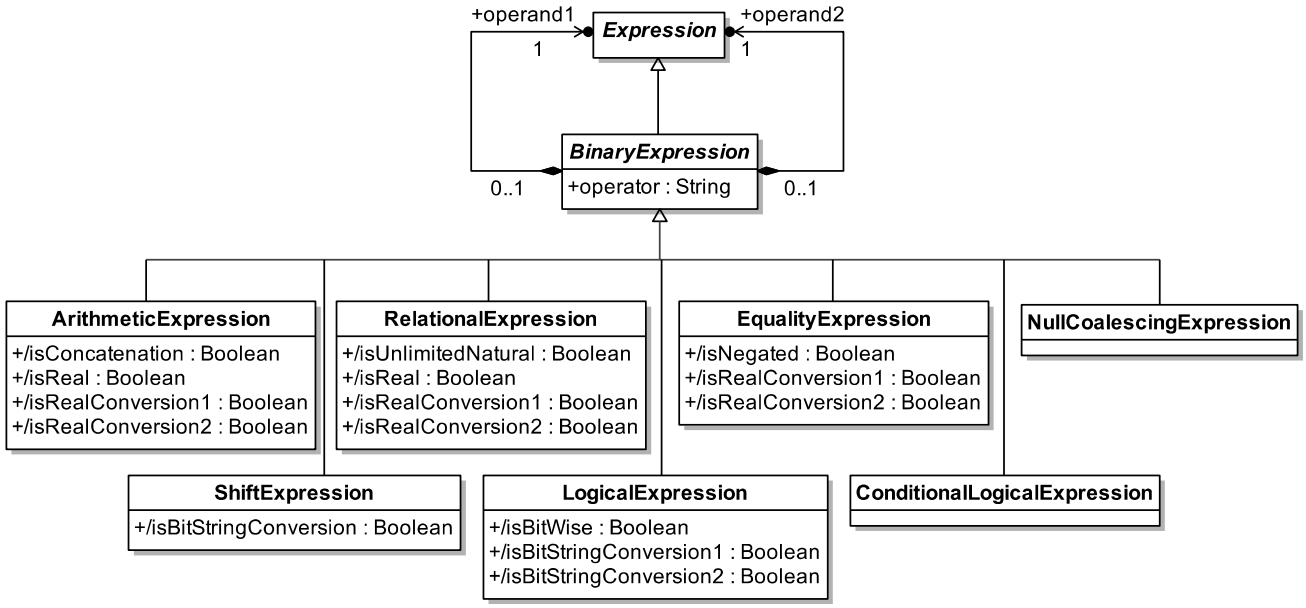

Figure 13.10 Binary Expressions

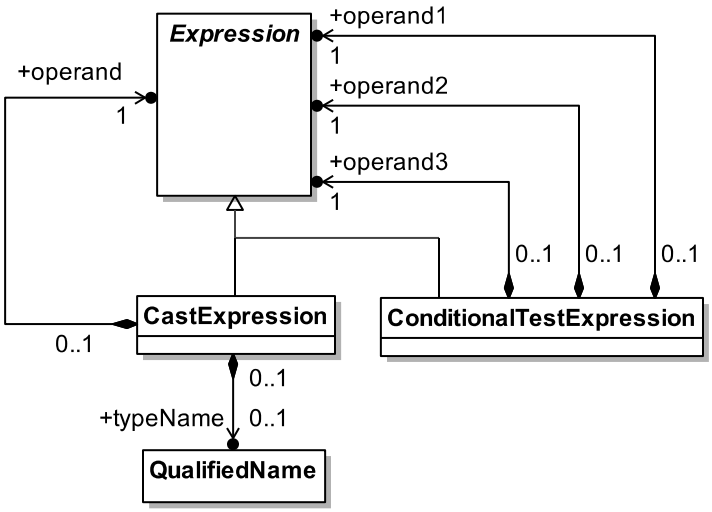

Figure 13.11 Cast and Conditional-Test Expressions

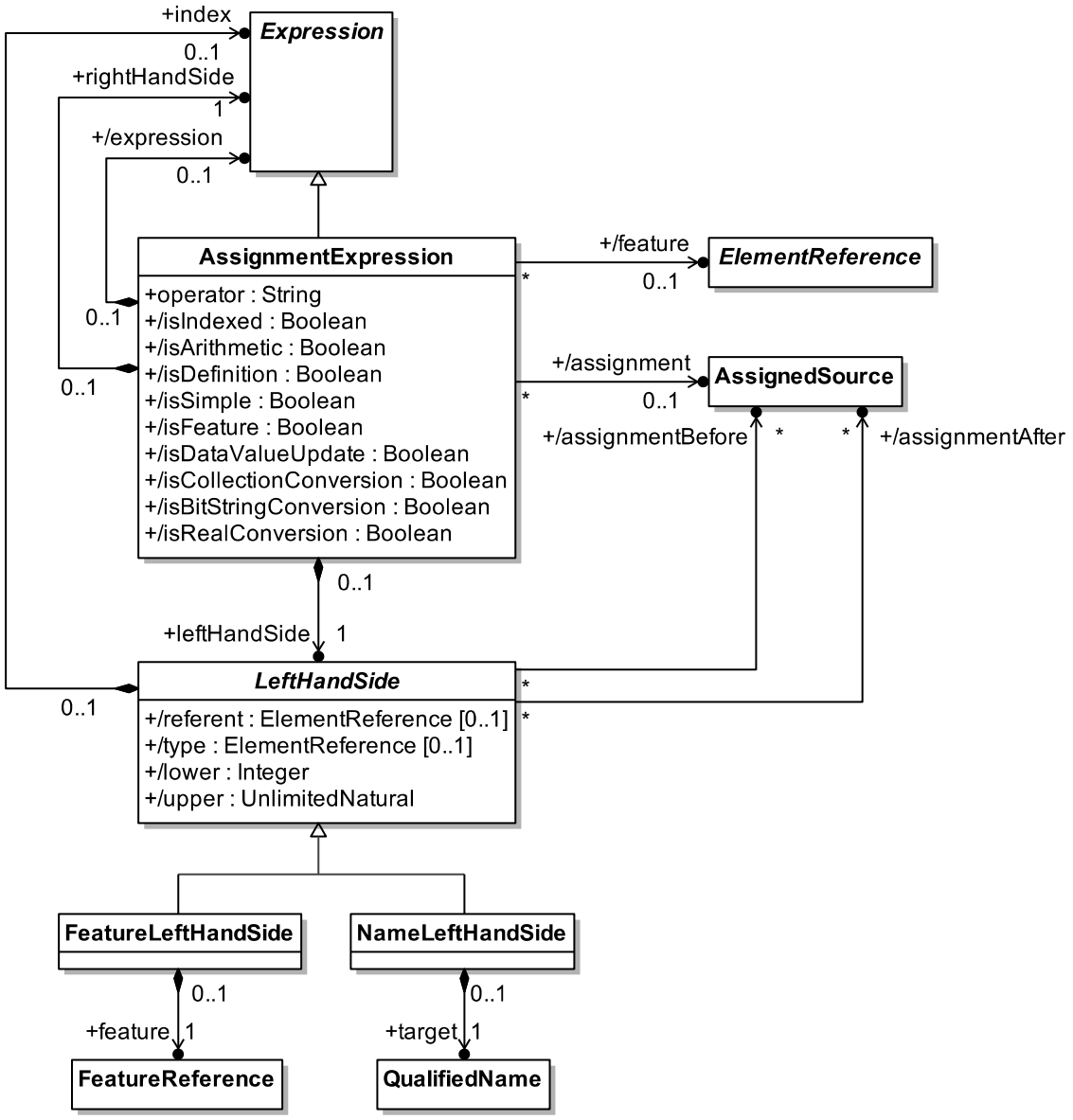

Figure 13.12 Assignment Expressions

13.2 Class Descriptions

13.2.1 ArithmeticExpression

A binary expression with an arithmetic operator.

Generalizations

- BinaryExpression

Synthesized Properties

None

Derived Properties

- isConcatenation : Boolean
  Whether this is a string concatenation expression.

- isReal : Boolean
  Whether this is a real arithmetic expression.

- isRealConversion1 : Boolean
  Whether Real conversion is required on the first operand of this expression.

- isRealConversion2 : Boolean
  Whether Real conversion is required on the second operand of this expression.

Constraints

[1] arithmeticExpressionIsConcatenationDerivation

An arithmetic expression is a string concatenation expression if its type is String.

[2] arithmeticExpressionIsRealConversion1Derivation

Real conversion is required if the type of an arithmetic expression is a type that conforms to type Real and the first operand expression has a type that conforms to type Integer.

[3] arithmeticExpressionIsRealConversion2Derivation

Real conversion is required if the type of an arithmetic expression is a type that conforms to type Real and the second operand expression has a type that conforms to type Integer.

[4] arithmeticExpressionIsRealDerivation

An arithmetic expression is a real computation if its type conforms to type Real.

[5] arithmeticExpressionLowerDerivation

An arithmetic expression has a multiplicity lower bound of 0 if its operator is / or if the lower bound if either operand expression is 0 and 1 otherwise.

[6] arithmeticExpressionOperandMultiplicity

The operand expressions of an arithmetic expressions must both have multiplicity upper bounds of 1.

[7] arithmeticExpressionOperandTypes

The operands of an arithmetic expression must both have types that conform to type Integer or Real, unless the operator is + or %. If the operator is +, then both operands may also have types that conform to type String. If the operator is %, then both operands must have types that conform to type Integer.

[8] arithmeticExpressionTypeDerivation

If both operands of an arithmetic expression operator are of a type that conforms to type Integer, then the type of the expression is Integer. If one operand is of a type that conforms to type Real and the other Integer or both are of a type that conforms to type Real, then the type of the expression is Real. If both operands are of a type that conforms to type String, then the type of the expression is String. Otherwise the expression has no type.

[9] arithmeticExpressionUpperDerivation

An arithmetic expression has a multiplicity upper bound of 1.

Helper Operations

[1] minLowerBound( ) : Integer

The minimum lower bound is 0 for operands of arithmetic expressions other than concatenations (this allows for the propagation of a null returned from a division by zero in an operand).

13.2.2 AssignmentExpression

An expression used to assign a value to a local name, parameter or property.

Generalizations

- Expression

Synthesized Properties

- leftHandSide : LeftHandSide

  The left-hand side of the assignment, to which a value is to be assigned.

- operator : String

  The image of the assignment operator used in the expression.

- rightHandSide : Expression

  The right-hand side expression of the assignment, which produces the value being assigned.

Derived Properties

- assignment : AssignedSource [0..1]

  If the left-hand side is a name, then the new assigned source for that name.

- expression : Expression [0..1]

  If this is a compound assignment, then the effective expression used to obtain the original value of the left-hand side to be updated.

- feature : ElementReference [0..1]

  If the left-hand side is a feature, then the referent for that feature.

- isArithmetic : Boolean

  If this is a compound assignment, whether the compound assignment operator is arithmetic or not.

- isBitStringConversion : Boolean

  Whether BitString conversion is required for this assignment.

- isCollectionConversion : Boolean

Whether collection conversion is required for this assignment.

- isDataValueUpdate : Boolean
  Whether this assignment updates an attribute of a data value held in a local name or parameter.

- isDefinition : Boolean
  Whether this assignment is the definition of a new local name or not.

- isFeature : Boolean
  Whether the left-hand side is a feature or not.

- isIndexed : Boolean
  If the left-hand side is a feature, whether it has an index or not.

- isRealConversion : Boolean
  Whether Real conversion is required for this assignment.

- isSimple : Boolean
  Whether this is a simple assignment or not.

Constraints

[1] assignmentExpressionAssignmentDerivation

The new assigned source for an assignment to a local name is the assignment expression (including a data value update). If the assignment is a definition, then the type is given by the right-side, otherwise the type is the same as for the previous assigned source for the local name. The multiplicity lower bound is 0 if the lower bound of the right-hand side is 0 and otherwise 1, and the multiplicity upper bound is 1 if the upper bound of the right-hand side is 1 and * otherwise, except that: if the left-hand side is a data-value update, the multiplicity is the same as for the previous assignment, and, if the left-hand side is indexed (but not a data-value update), the multiplicity is [0..*]. If the assignment expression does not require any conversions, then the subtype of the assignment is the type of the right-hand side expression; otherwise it is null.

[2] assignmentExpressionAssignmentsBefore

The assigned source of a name before the right-hand side expression of an assignment expression is the same as the assigned source before the assignment expression. The assigned source of a name before the left-hand side is the assigned source after the right-hand side expression.

[3] assignmentExpressionCompoundAssignmentMultiplicityConformance

For a compound assignment, both the left-hand and right-hand sides must have a multiplicity upper bound of 1.

[4] assignmentExpressionCompoundAssignmentTypeConformance

For a compound assignment, if the operator is an arithmetic operator, then either the left-hand side and the right-hand side both have types that conform to type Integer, the left-hand side has a type that conforms to type Real and the right-hand side has a type that conforms to type Integer or Real, or they

both have types that conform to type String and the operator is +. If the operator is a logical operator, then either the left-hand side and the right-hand side both have types that conform to type Boolean or Bit String or the left-hand side has a type that conforms to type Bit String and the right-hand side has a type that conforms to type Integer. If the operator is a shift operator, then the left-hand side must have a type that conforms to type Bit String and the right-hand side must have a type that conforms to type Integer.

[5] assignmentExpressionDataValueUpdateLegality

If an assignment expression has a feature with a primary expression whose type is a data type, then the assignment expression must be a data value update.

[6] assignmentExpressionExpressionDerivation

For a compound assignment, the effective expression is the left-hand side treated as a name expression, property access expression or sequence access expression, as appropriate for evaluation to obtain the original value to be updated.

[7] assignmentExpressionFeatureDerivation

If the left-hand side of an assignment expression is a feature, then the feature of the assignment is the referent of the left-hand side.

[8] assignmentExpressionIsArithmeticDerivation

An assignment expression is an arithmetic assignment if its operator is a compound assignment operator for an arithmetic operation.

[9] assignmentExpressionIsBitStringConversionDerivation

An assignment requires BitString conversion if the type of the left-hand side is BitString and either the type of the right-hand side is Integer or collection conversion is required and the type of the right-hand side is a collection class whose sequence type is Integer.

[10] assignmentExpressionIsCollectionConversionDerivation

An assignment requires collection conversion if the type of the right-hand side is a collection class and its multiplicity upper bound is 1, and the type of the left-hand side is not a collection class.

[11] assignmentExpressionIsDataValueUpdateDerivation

An assignment expression is a data value update if its left hand side is an attribute of a data value held in a local name or parameter.

[12] assignmentExpressionIsDefinitionDerivation

An assignment expression is a definition if it is a simple assignment and its left hand side is a local name for which there is no assignment before the expression.

[13] assignmentExpressionIsFeatureDerivation

The left hand side of an assignment expression is a feature if it is a a feature left-hand side or a name left-hand side for a name that disambiguates to a feature.

[14] assignmentExpressionIsIndexedDerivation

The left hand side of an assignment expression is indexed if it has an index.

[15] assignmentExpressionIsRealConversionDerivation

An assignment requires Real conversion if the type of the left-hand side is a type that conforms to type Real and either the type of the right-hand side is a type that conforms to type Integer or collection conversion is required and the type of the right-hand side is a collection class whose sequence type is a type that conforms to type Integer.

[16] assignmentExpressionIsSimpleDerivation

An assignment expression is a simple assignment if the assignment operator is "=".

[17] assignmentExpressionLowerDerivation

A simple assignment expression has the same multiplicity lower bound as its right-hand side expression. A compound assignment expression has the same multiplicity as its left-hand side.

[18] assignmentExpressionSimpleAssignmentMultiplicityConformance

If the left-hand side of a simple assignment is not a local name and the multiplicity lower bound of the left-hand side is greater than 0, then the multiplicity lower bound of the right-hand side cannot be 0. If the left-hand side is not a new local name and the multiplicity upper bound of the left-hand side is less than or equal to 1, then the multiplicity upper bound of the right-hand side cannot be greater than that of the left-hand side.

[19] assignmentExpressionSimpleAssignmentTypeConformance

If the left-hand side of a simple assignment is not a new local name, and the right-hand side is not null, then either the left-hand side must be untyped or the right-hand side expression must have a type that conforms to the type of the left-hand side.

[20] assignmentExpressionTypeDerivation

A simple assignment expression has the same type as its right-hand side expression. A compound assignment expression has the same type as its left-hand side.

[21] assignmentExpressionUpperDerivation

An assignment expression has the same multiplicity upper bound as its right-hand side expression.

Helper Operations

[1] adjustMultiplicity(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the left-hand side is not indexed and is not a feature reference, then the assigned name is considered be known null if the condition is true, or known non-null if the condition is false. The right-hand side is then also checked for known nulls or non-nulls.

[2] adjustType(assignments : AssignedSource [*], subtype : ElementReference) : AssignedSource [*]

If the left-hand side is not indexed and is not a feature reference, then the assigned name is considered to have the given subtype. The type of the right-hand side is then also adjusted as appropriate.

[3] declaredType() : ElementReference

If an assignment expression is a simple assignment, then its declared type is the declared type of the right-hand side expression. Otherwise it is the type of the assignment expression.

[4] updateAssignments ( ) : AssignedSource [*]

The assignments after an assignment expression are the assignments after the left-hand side, updated by the assignment from the assignment statement, if any.

13.2.3 BehaviorInvocationExpression

An invocation of a behavior referenced by name.

Generalizations

- InvocationExpression

Synthesized Properties

- target : QualifiedName

  The qualified name of the behavior to be invoked.

Derived Properties

None

Constraints

[1] behaviorInvocationExpressionAlternativeConstructor

The referent may only be a constructor (as a result of the target disambiguating to a feature reference) if this behavior invocation expression is the expression of an expression statement that is the first statement in the definition for the method of a constructor operation.

[2] behaviorInvocationExpressionArgumentCompatibility

If the target qualified name does not disambiguate to a feature reference, then each input argument expression must be assignable to its corresponding parameter and each output argument expression must be assignable from its corresponding parameter. (Note that this implies that the type of an argument expression for an inout parameter must be the same as the type of that parameter.)

[3] behaviorInvocationExpressionFeatureDerivation

If the target qualified name disambiguates to a feature reference, then the feature of a behavior invocation expression is that feature reference.

[4] behaviorInvocationExpressionReferentConstraint

If the target qualified name does not disambiguate to a feature reference, then it must resolve to a behavior or an association end, and, if it is a template behavior, then the implicit binding of this template must be legal. Otherwise it must resolve to a single feature referent according to the overloading resolution rules, unless it is an implicit destructor call (in which case it has no referent).

[5] behaviorInvocationExpressionReferentDerivation

If the target of a behavior invocation expression resolves to a behavior, then the referent of the expression is that behavior. If the target disambiguates to a feature reference, then the reference is the

operation or signal being invoked. Otherwise, if the target resolves to a property that is an association end, then the referent is that property.

Helper Operations

[1] adjustAssignments(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the invoked behavior is CollectionFunctions::isEmpty or SequenceFunctions::IsEmpty, then check the argument expression for known nulls and non-nulls using the given truth condition. If the invoked behavior is CollectionFunctions::notEmpty or SequenceFunctions::NotEmpty, then check the argument expression for known nulls and non-nulls using the negation of the given truth condition.

13.2.4 BinaryExpression

An expression consisting of an operator acting on two operand expressions.

Generalizations

- Expression

Synthesized Properties

- operand1 : Expression
  The expression giving the first operand.

- operand2 : Expression
  The expression giving the second operand.

- operator : String
  The symbol representing the operator.

Derived Properties

None

Constraints

[1] binaryExpressionOperandAssignments

The assignments in the operand expressions of a binary expression must be valid (as determined by the validateAssignments helper operation).

[2] binaryExpressionOperandMultiplicity

The operands of a binary expression must both have a multiplicity lower bound no less than that given by the minLowerBound helper operation. The operands of a binary expression must both have a multiplicity upper bound no greater than that given by the maxUpperBound helper operation.

Helper Operations

[1] minLowerBound( ) : Integer
By default, the minimum allowed lower bound for an operand of a binary expression is 1.

[2] maxUpperBound( ) : UnlimitedNatural
By default, the maximum allowed upper bound for an operand of a binary expression is 1.

[3] updateAssignments ( ) : AssignedSource [*]

The assignments after a binary expression include all the assignments before the expression that are not reassigned in either operand expression, plus the new assignments from each of the operand expressions.

[4] validateAssignments ( ) : Boolean

In general the assignments before the operand expressions of a binary expression are the same as those before the binary expression and, if an assignment for a name is changed in one operand expression, then the assignment for that name may not change in the other operand expression. (This is overridden for conditional logical expressions.)

13.2.5 BitStringUnaryExpression

Generalizations

- UnaryExpression

Synthesized Properties

None

Derived Properties

- isBitStringConversion : Boolean

  Whether BitString conversion is required on the operand expression.

Constraints

[1] bitStringUnaryExpressionIsBitStringConversionDerivation

BitString conversion is required if the operand expression of a BitString unary expression has a type that conforms to type Integer.

[2] bitStringUnaryExpressionLowerDerivation

A BitString unary expression has a multiplicity lower bound of 1.

[3] bitStringUnaryExpressionOperand

The operand expression of a BitString unary expression must have a type that conforms to type BitString or Integer and multiplicity lower and upper bounds of 1.

[4] bitStringUnaryExpressionTypeDerivation

A BitString unary expression has type BitString.

[5] bitStringUnaryExpressionUpperDerivation

A BitString unary expression has a multiplicity upper bound of 1.

Helper Operations

None

13.2.6 BooleanLiteralExpression

An expression that comprises a Boolean literal.

Generalizations

- LiteralExpression

Synthesized Properties

- image : String

  The textual image of the literal token for this expression.

Derived Properties

None

Constraints

[1] booleanLiteralExpressionTypeDerivation

The type of a boolean literal expression is Boolean.

Helper Operations

None

13.2.7 BooleanUnaryExpression

A unary expression with a Boolean operator.

Generalizations

- UnaryExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] booleanUnaryExpressionLowerDerivation

A Boolean unary expression has the same multiplicity lower bound as its operand expression.

[2] booleanUnaryExpressionOperand

The operand expression of a Boolean unary expression must have a type that conforms to type Boolean and a multiplicity upper bound of 1.

[3] booleanUnaryExpressionTypeDerivation

A Boolean unary expression has type Boolean.

[4] booleanUnaryExpressionUpperDerivation

A Boolean unary expression has a multiplicity upper bound of 1.

Helper Operations

[1] adjustAssignments(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the expression is a negation, then check the operand expression for known nulls and non-nulls based on the negation of the given truth condition.

13.2.8 CastExpression

An expression used to filter values by type.

Generalizations

- Expression

Synthesized Properties

- operand : Expression

  The operand expression of the cast expression.

- typeName : QualifiedName [0..1]

  The named type of the cast expression (if any)

Derived Properties

None

Constraints

[1] castExpressionAssignmentsBefore

The assignments before the operand of a cast expression are the same as those before the cast expression.

[2] castExpressionLowerDerivation

If the type of a cast expression is empty, or its type conforms to Integer and the type of its operand expression conforms to BitString or Real, or its type conforms to BitString or Real and its operand's type conforms to Integer, or its operand's type conforms to its type, then the multiplicity lower bound of the cast expression is the same as that of its operand expression. Otherwise it is 0.

[3] castExpressionTypeDerivation

The type of a cast expression is the referent of the given type name (if there is one).

[4] castExpressionTypeResolution

If the cast expression has a type name, then it must resolve to a non-template classifier.

[5] castExpressionUpperDerivation

A cast expression has a multiplicity upper bound that is the same as the upper bound of its operand expression.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

The assignments after a cast expression are the same as those after its operand expression.

13.2.9 ClassExtentExpression

An expression used to obtain the objects in the extent of a class.

Generalizations

- Expression

Synthesized Properties

- className : QualifiedName

  The name of the class whose extent is to be obtained.

Derived Properties

None

Constraints

[1] classExtentExpressionExtentType

The given type name must resolve to a non-template class.

[2] classExtentExpressionLowerDerivation

The multiplicity lower bound of a class extent expression is 0.

[3] classExtentExpressionTypeDerivation

The type of a class extent expression is the given class.

[4] classExtentExpressionUpperDerivation

The multiplicity upper bound of a class expression is *.

Helper Operations

None

13.2.10 ClassificationExpression

An expression used to test the dynamic type of its operand.

Generalizations

- UnaryExpression

Synthesized Properties

- typeName : QualifiedName

  The name of the type that the operand is to be tested against.

Derived Properties

- isDirect : Boolean

  Whether the test is for the operand to have the given type directly or to only conform to the given type.

- referent : ElementReference
  Whether the test is for the operand to have the given type directly or to only conform to the given type.

Constraints

[1] classificationExpressionIsDirectDerivation

A classification expression is direct if its operator is "hastype".

[2] classificationExpressionLowerDerivation

A classification expression has a multiplicity lower bound of 1.

[3] classificationExpressionOperand

The operand expression of a classification expression must have multiplicity lower and upper bounds of 1.

[4] classificationExpressionReferentDerivation

The referent of a classification expression is the classifier to which the type name resolves.

[5] classificationExpressionTypeDerivation

A classification expression has type Boolean.

[6] classificationExpressionTypeName

The type name in a classification expression must resolve to a non-template classifier.

[7] classificationExpressionUpperDerivation

A classification expression has a multiplicity upper bound of 1.

Helper Operations

[1] adjustAssignments(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the truth condition is true and the type of the operand of a classification expression does not conform to the referent type of the classification expression, then set the known type of the operand of the classification expression to be the referent type of the classification expression.

13.2.11 CollectOrIterateExpression

A sequence expansion expression with a collect or iterate operation.

Generalizations

- SequenceExpansionExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] collectOrIterateExpressionLowerDerivation

A collect or iterate expression has a multiplicity lower bound that is the product of the bounds of its primary and argument expressions.

[2] collectOrIterateExpressionTypeDerivation

A collect or iterate expression has the same type as its argument expression.

[3] collectOrIterateExpressionUpperDerivation

A collect or iterate expression has a multiplicity upper bound that is the product of the bounds of its primary and argument expressions.

Helper Operations

None

13.2.12 ConditionalLogicalExpression

A binary expression with a conditional logical expression, for which the evaluation of the second operand expression is conditioned on the result of evaluating the first operand expression.

Generalizations

- BinaryExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] conditionalLogicalExpressionLowerDerivation

A conditional logical expression has a multiplicity lower bound of 1.

[2] conditionalLogicalExpressionOperands

The operands of a conditional logical expression must have a type that conforms to type Boolean.

[3] conditionalLogicalExpressionTypeDerivation

A conditional logical expression has type Boolean.

[4] conditionalLogicalExpressionUpperDerivation

A conditional logical expression has a multiplicity upper bound of 1.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

If a name has the same assigned source after the second operand expression as before it, then that is its assigned source after the conditional logical expression. Otherwise its assigned source after the

conditional logical expression is the conditional logical expression itself. If a name is unassigned before the second operand expression but assigned after it, then it has a multiplicity lower bound of 0 after the conditional logical expression.

[2] validateAssignments ( ) : Boolean

The assignments before the first operand expression of a conditional logical expression are the same as those before the conditional logical expression. The assignments before the second operand expression are the same as those after the first operand expression, adjusted for known nulls and non-nulls based on the first operand expression being true, for a conditional-and expression, or false, for a conditional-or expression.

13.2.13 ConditionalTestExpression

An expression that uses the value of one operand expression to condition the evaluation of one of two other operand expressions.

Generalizations

- Expression

Synthesized Properties

- operand1 : Expression

  The first operand expression, which provides the condition to be tested.

- operand2 : Expression

  The second operand expression, to be evaluated if the condition is true.

- operand3 : Expression

  The third operand expression, to be evaluated if the condition is false.

Derived Properties

None

Constraints

[1] conditionalTestExpressionAssignmentsBefore

The assignments before the first operand expression of a conditional-test expression are the same as those before the conditional-test expression. The assignments before the second and third operand expressions are the same as those after the first operand expression.

[2] conditionalTestExpressionCondition

The first operand expression of a conditional-test expression must be of a type that conforms to type Boolean and have multiplicity lower and upper bounds of 1.

[3] conditionalTestExpressionLowerDerivation

The multiplicity lower bound of a conditional-test operator expression is the minimum of the multiplicity lower bounds of its second and third operand expressions.

[4] conditionalTestExpressionTypeDerivation

The type of a conditional-test operator expression is the effective common ancestor (if one exists) of the types of its second and third operand expressions.

[5] conditionalTestExpressionUpperDerivation

The multiplicity upper bound of a conditional-test operator expression is the maximum of the multiplicity upper bounds of its second and third operand expressions.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

Returns unchanged all assignments for local names that are not reassigned in either the second or third operand expressions. Any local names that have different assignments after the second and third operand expressions are adjusted to have the conditional-test expression as their assigned source. If such a local name is defined in one operand expression but not the other, then it is adjusted to have multiplicity lower bound of 0 after the conditional test expression. If a local name has a new assignment after each of the second and third expressions, then, after the conditional-test expression, it has a type that is the effective common ancestor of its type after the second and third operand expressions, adjusted for known null and non-null names from the first operand expression being true, for the second operand expression, or false, for the third operand expression, a multiplicity lower bound that is the minimum of the lower bounds after the second and third operand expressions and a multiplicity upper bound that is the maximum of the upper bounds after the second and third expressions.

13.2.14 EqualityExpression

A binary expression that tests the equality of its operands.

Generalizations

- BinaryExpression

Synthesized Properties

None

Derived Properties

- isNegated : Boolean

  Whether the test is for being not equal.

- isRealConversion1 : Boolean

  Whether Real conversion is required on the first operand of this expression.

- isRealConversion2 : Boolean

  Whether Real conversion is required on the second operand of this expression.

Constraints

[1] equalityExpressionIsNegatedDerivation

An equality expression is negated if its operator is "!=".

[2] arithmeticExpressionIsRealConversion1Derivation
An equality expression requires Real conversion if the first operand is of type Integer and the second is of type Real.

[3] arithmeticExpressionIsRealConversion2Derivation
An equality expression requires Real conversion if the first operand is of type Real and the second is of type Integer.

[3] equalityExpressionLowerDerivation
An equality expression has a multiplicity lower bound of 1.

[4] equalityExpressionTypeDerivation
An equality expression has type Boolean.

[5] equalityExpressionUpperDerivation
An equality expression has a multiplicity upper bound of 1.

Helper Operations

[1] adjustAssignments (assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]
If the one operand expression has multiplicity 0..0, then check the other operand expression for known nulls and non-nulls, using the exclusive-or of the given truth condition and whether the equality expression is negated or not.

[2] minLowerBound ( ) : Integer
The minimum lower bound is 0 for operands of equality expressions.

13.2.15 Expression

A model of the common properties derived for any Alf expression.

NOTE: The derivations for all properties of Expression except AssignmentsAfter are specific to its various subclasses.

Generalizations

- SyntaxElement

Synthesized Properties

None

Derived Properties

- assignmentAfter : AssignedSource [*]

  The assigned sources for local names available lexically after this expression. This includes not only any assignments made within the expression, but also any assignments that are unchanged from before the expression.

- assignmentBefore : AssignedSource [*]
  The assigned sources for local names available lexically before this expression.

- lower : Integer
  The statically determined lower bound of the multiplicity of this expression.

- type : ElementReference [0..1]
  A reference to the element that specifies the statically determined type for this expression (if any).

- upper : UnlimitedNatural
  The statically determined upper bound of the multiplicity of this expression.

Constraints

[1] expressionAssignmentAfterDerivation

The assignments after an expression are given by the result of the updateAssignments helper operation.

[2] expressionUniqueAssignments

No name may be assigned more than once before or after an expression.

Helper Operations

[1] adjustAssignments (assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

Returns the given assignments, adjusted for known nulls, known non-nulls and best known types, based on the given truth condition. By default, no changes are made. (This operation is overridden for conditional logical, binary unary, equality, behavior invocation, sequence operation and classification expressions that may be used to form checks for null and non-null values and type classification.)

[2] adjustMultiplicity (assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

Returns the given assignments, adjusted for known nulls and non-nulls, based on the given truth condition. By default, no changes are made. (This operation is overridden for name and assignment expressions that may be used to provide the names that are checked for being null or non-null.)

[3] adjustType (assignments : AssignedSource [*], subtype : ElementReference) : AssignedSource [*]

Returns the given assignments, adjusted for the given best-known subtype. By default, no changes are made. (This operation is overridden by name and assignment expressions that may be used to provide the names that are checked for type classification.)

[4] declaredType ( ) : ElementReference

Return the type of the expression, based on the originally declared types of names in the expression. By default, this is the expression type.

[5] updateAssignments ( ) : AssignedSource [*]

Returns the assignments from before this expression updated for any assignments made in the expression. By default, this is the same set as the assignments before the expression. This operation is redefined only in subclasses of Expression for kinds of expressions that make assignments.

13.2.16 ExtentOrExpression

The target of a sequence operation, reduction or expansion expression, which may be either a primary expression or a class name denoting the class extent.

Generalizations

None

Synthesized Properties

- name : QualifiedName [0..1]

  If the target is a qualified name, then that name, before it is disambiguated into either a name expression or a class name.

- nonNameExpression : Expression [0..1]

  The target primary expression, if it is not a qualified name.

Derived Properties

- expression : Expression

  The effective expression for the target.

Constraints

[1] extentOrExpressionExpressionDerivation

The effective expression for the target is the parsed primary expression, if the target is not a qualified name, a name expression, if the target is a qualified name other than a class name, or a class extent expression, if the target is the qualified name of a class.

Helper Operations

None

13.2.17 FeatureInvocationExpression

An invocation of a feature referenced on a sequence of instances.

Generalizations

- InvocationExpression

Synthesized Properties

- target : FeatureReference [0..1]

  A feature reference to the target feature to be invoked.

Derived Properties

None

Constraints

[1] featureInvocationExpressionAlternativeConstructor

An alternative constructor invocation may only occur in an expression statement as the first statement in the definition for the method of a constructor operation.

[2] featureInvocationExpressionFeatureDerivation

If a feature invocation expression has an explicit target, then that is its feature. Otherwise, it is an alternative constructor call with its feature determined implicitly.

[3] featureInvocationExpressionImplicitAlternativeConstructor

If there is no target feature expression, then the implicit feature with the same name as the target type must be a constructor.

[4] featureInvocationExpressionReferentDerivation

If a feature invocation expression is an implicit object destruction, it has no referent. Otherwise, its referent is the operation or signal being invoked.

[5] featureInvocationExpressionReferentExists

If a feature invocation expression is not an implicit destructor call, then it must be possible to determine a single valid referent for it according to the overloading resolution rules.

Helper Operations

None

13.2.18 FeatureLeftHandSide

A left-hand side that is a property reference.

Generalizations

- LeftHandSide

Synthesized Properties

- feature : FeatureReference

  The structural feature being assigned.

Derived Properties

None

Constraints

[1] featureLeftHandSideAssignmentAfterDerivation

The assignments after a feature left-hand side are the assignments after the expression of the feature reference or, if there is an index, those after the index expression.

[2] featureLeftHandSideAssignmentBeforeDerivation

The assignments before the expression of the feature reference of a feature left-hand side are the assignments before the feature left-hand side.

[3] featureLeftHandSideAssignmentsBefore

If a feature left-hand side has an index, then the assignments before the index expression are the assignments after the expression of the feature reference.

[4] featureLeftHandSideFeatureExpression

The expression of the feature reference of a feature left-hand side must have a multiplicity upper bound of 1.

[5] featureLeftHandSideIndexedFeature

If a feature left-hand side has an index, then the referent of the feature must be ordered and non-unique.

[6] featureLeftHandSideLowerDerivation

If a feature left-hand side is indexed, then its lower bound is 0. Otherwise, its lower bound is that of its referent.

[7] featureLeftHandSideReferentConstraint

The feature of a feature-left hand side must have a single referent that is a structural feature.

[8] featureLeftHandSideReferentDerivation

The referent of a feature left-hand side is the structural feature to which the feature reference of the left-hand side resolves.

[9] featureLeftHandSideTypeDerivation

The type of a feature left-hand side is the type of its referent.

[10] featureLeftHandSideUpperDerivation

If a feature left-hand side is indexed, then its upper bound is 1. Otherwise, its upper bound is that of its referent.

Helper Operations

None

13.2.19 FeatureReference

A reference to a structural or behavioral feature of the type of its target expression or a binary association end the opposite end of which is typed by the type of its target expression.

Generalizations

- SyntaxElement

Synthesized Properties

- expression : Expression

  The target expression.

- nameBinding : NameBinding

  The name of the feature.

Derived Properties

- referent : ElementReference [*]

  The features referenced by this feature reference.

Constraints

[1] featureReferenceReferentDerivation

The features referenced by a feature reference include the features of the type of the target expression and the association ends of any binary associations whose opposite ends are typed by the type of the target expression.

[2] featureReferenceTargetType

The target expression of the feature reference may not be untyped, nor may it have a primitive or enumeration type.

Helper Operations

None

13.2.20 ForAllOrExistsOrOneExpression

A sequence expansion expression with a forAll, exists or one operation.

Generalizations

- SequenceExpansionExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] forAllOrExistsOrOneExpressionArgument

The argument of a forAll, exists or one expression must have a type that conforms to type Boolean and a multiplicity upper bound of 1.

[2] forAllOrExistsOrOneExpressionLowerDerivation

A forAll, exists or one expression has a multiplicity lower bound of 1.

[3] forAllOrExistsOrOneExpressionTypeDerivation

A forAll, exists or one expression has the type Boolean.

[4] forAllOrExistsOrOneExpressionUpperDerivation

A forAll, exists or one expression has a multiplicity upper bound of 1.

Helper Operations

None

13.2.21 IncrementOrDecrementExpression

A unary expression with either an increment or decrement operator.

Generalizations

- Expression

Synthesized Properties

- operator : String
  The operator for this increment or decrement expression, either "++" for increment or "--" for decrement.

- isPrefix : Boolean = false
  Whether the operator is being used as a prefix or a postfix.

- operand : LeftHandSide
  The operand, which must have the form of an assignment left-hand side.

Derived Properties

- assignment : AssignedSource [0..1]
  If the operand is a name, then the new assigned source for that name.

- expression : Expression
  The effective expression used to obtain the original value of the operand to be updated.

- feature : ElementReference [0..1]
  If the operand is a feature, then the referent for that feature.

- isDataValueUpdate : Boolean
  Whether this expression updates an attribute of a data value held in a local name or parameter.

- isFeature : Boolean
  Whether the operand is a feature or not.

- isIndexed : Boolean
  If the operand is a feature, whether it has an index or not.

Constraints

[1] incrementOrDecrementExpressionAssignmentDerivation

If the operand of an increment or decrement expression is a name, then the assignment for the expression is a new assigned source for the name with the expression as the source.

[2] incrementOrDecrementExpressionAssignmentsBefore

The assignments before the operand of an increment or decrement expression are the same as those before the increment or decrement expression.

[3] incrementOrDecrementExpressionExpressionDerivation

The effective expression for the operand of an increment or decrement expression is the operand treated as a name expression, property access expression or sequence access expression, as appropriate for evaluation to obtain the original value to be updated.

[4] incrementOrDecrementExpressionFeatureDerivation

If the operand of an increment or decrement expression is a feature, then the referent for the operand.

[5] incrementOrDecrementExpressionIsDataValueUpdateDerivation

An increment or decrement expression is a data value update if its operand is an attribute of a data value held in a local name or parameter.

[6] incrementOrDecrementExpressionIsFeatureDerivation

An increment or decrement expression has a feature as its operand if the operand is a feature left-hand side or a name left-hand side for a name that disambiguates to a feature.

[7] incrementOrDecrementExpressionIsIndexedDerivation

An increment or decrement expression is indexed if its operand is indexed.

[8] incrementOrDecrementExpressionLowerDerivation

An increment or decrement expression has the same multiplicity lower bound as its operand expression.

[9] incrementOrDecrementExpressionOperand

The operand expression must have a type that conforms to type Integer or Real and a multiplicity upper bound of 1.

[10] incrementOrDecrementExpressionTypeDerivation

If the operand of an increment or decrement expression is of a type that conforms to type Integer or Real, then the type of the expression is Integer or Real, respectively. Otherwise the expression has no type.

[11] incrementOrDecrementExpressionUpperDerivation

An increment or decrement expression has a multiplicity upper bound of 1.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

The assignments after an increment and decrement expression include all those after its operand expression. Further, if the operand expression, considered as a left hand side, is a local name, then this is reassigned.

13.2.22 InstanceCreationExpression

An expression used to create a new instance of a class or data type.

Generalizations

- InvocationExpression

Synthesized Properties

- constructor : QualifiedName [0..1]

  The name of the class constructor operation to be invoked or the name of a class or data type.

Derived Properties

- isConstructorless : Boolean

  Whether this is a constructorless object creation expression.

- isObjectCreation : Boolean

  Whether this is an object creation expression or a data value creation expression.

Constraints

[1] instanceCreationExpressionConstructor

The constructor name must resolve to a constructor operation (that is compatible with the tuple argument expressions), a class or a data type, but not both a class and a data type. Further, if the constructor name of an instance creation expression is empty, then the referent must be determined from the context of use of the expression.

[2] instanceCreationExpressionConstructorlessLegality

If the expression is constructorless, then its tuple must be empty and the referent class must not have any owned operations that are constructors.

[3] instanceCreationExpressionDataTypeCompatibility

If an instance creation expression is a data value creation (not an object creation), then the tuple argument expressions are matched with the attributes of the named type.

[4] instanceCreationExpressionFeatureDerivation

There is no feature for an instance creation expression.

[5] instanceCreationExpressionIsConstructorlessDerivation

An instance creation expression is constructorless if its referent is a class.

[6] instanceCreationExpressionIsObjectCreationDerivation

An instance creation expression is an object creation if its referent is not a data type.

[7] instanceCreationExpressionReferent

If the referent of an instance creation expression is an operation, then the class of that operation must not be abstract. Otherwise, the referent is a class or data type, which must not be abstract.

[8] instanceCreationExpressionReferentDerivation

The referent of an instance creation expression is normally the constructor operation, class or data type to which the constructor name resolves. However, if the referent is an operation whose class is abstract or is a class that is itself abstract, and there is an associated Impl class constructor, then the referent is

the Impl class constructor.

Helper Operations

[1] parameterElements ( ) : ElementReference [*]

Returns the parameters of a constructor operation or the attributes of a data type, or an empty set for a constructorless instance creation.

13.2.23 InvocationExpression

An expression denoting the invocation of a behavior or operation, or the sending of a signal.

Generalizations

- Expression

Synthesized Properties

- tuple : Tuple

  The tuple for the invocation expression.

Derived Properties

- boundReferent : ElementReference [0..1]

  If the referent of the invocation expression is a template behavior, then the implicit bound element for the referent, otherwise the same as the referent.

- feature : FeatureReference [0..1]

  For an invocation of a feature, the reference to that feature. This property is set for a feature invocation expression or for an expression initially parsed as a behavior invocation expression that disambiguates to a feature invocation expression.

- isAssociationEnd : Boolean

  Whether this is an association read or not.

- isBehavior : Boolean

  Whether this is a behavior invocation or not.

- isDestructor : Boolean

  If this is an operation call, whether the operation is a destructor.

- isImplicit : Boolean

  Whether this is an implicit object destruction.

- isOperation : Boolean

  Whether this is an operation call or not.

- isSignal : Boolean

  Whether this is a signal send or not.

- parameter : ElementReference [*]
  Element references to the parameters of the referent, in order.

- referent : ElementReference [0..1]
  The behavior, operation or signal being invoked. The derivation of this property is specific to each kind of invocation expression.

Constraints

[1] invocationExpressionAssignmentAfter

If the invocation is a sequence feature invocation, then the assignments after the tuple of the invocation expression must be the same as the assignments before.

[2] invocationExpressionAssignmentsBefore

The assignments before the target expression of the feature reference of an invocation expression (if any) are the same as the assignments before the invocation expression.

[3] invocationExpressionIsAssociationEndDerivation

An invocation expression is an association end read if its referent is an association end.

[4] invocationExpressionIsBehaviorDerivation

An invocation expression is a behavior invocation if its referent is a behavior.

[5] invocationExpressionBoundReferentDerivation

If the referent of an invocation expression is a template behavior, then the bound referent is the implicit template binding of this template; otherwise it is the same as the referent. For an implicit template binding, the type arguments of for the template are inferred from the types of the arguments for in and inout parameters of the template behavior. If the resulting implicit template binding would not be a legal binding of the template behavior, then the invocation expression has no bound referent.

[6] invocationExpressionIsDestructorDerivation

An invocation expression is a destructor call either implicitly or if it is an explicit operation call to a destructor operation.

[8] invocationExpressionIsImplicitDerivation

An invocation expression is an implicit object destruction if it has a feature with the name "destroy" and no explicit referents.

[7] invocationExpressionIsOperationDerivation

An invocation expression is an operation call if its referent is an operation.

[9] invocationExpressionIsSignalDerivation

An invocation expression is a signal send if its referent is a signal.

[10] invocationExpressionLowerDerivation

If the referent of an invocationExpression is an operation or behavior with a return parameter, then the lower bound of the expression is that of the return parameter. If the referent is a classifier, then the lower bound is 1. If the referent is a property, then the lower bound is that of the property. Otherwise the lower bound is 0.

[11] invocationExpressionParameterDerivation

The parameters of an invocation expression are given by the result of the parameterElements helper operation.

[12] invocationExpressionTemplateParameters

If the referent of the invocation expression is a template, then all of its template parameters must be classifier template parameters. Note: This allows for the possibility that the referent is not an Alf unit, in which case it could have non-classifier template parameters.

[13] invocationExpressionTypeDerivation

If the (bound) referent of an invocationExpression is an operation or behavior with a return parameter, then the type of the expression is that of the return parameter (if any). If the referent is a classifier, then the type is the referent. If the referent is a property, then the type is that of the property. Otherwise the expression has no type.

[14] invocationExpressionUpperDerivation

If the referent of an invocationExpression is an operation or behavior with a return parameter, then the upper bound of the expression is that of the return parameter. If the referent is a classifier, then the upper bound is 1. If the referent is a property, then the upper bound is that of the property. Otherwise the upper bound is 0.

Helper Operations

[1] parameterElements ( ) : ElementReference [*] {ordered}

Returns references to the elements that act as the parameters of the referent. If the referent is a behavior or operation, these are the owned parameters, in order. If the (bound) referent is an association end, then the parameters are the other association ends of the association of the referent end, which are treated as if they were in parameters. Otherwise (by default), they are any properties of the referent (e.g., signal attributes), which are treated as if they were in parameters. (This is defined as a helper operation, so that it can be overridden by subclasses of InvocationExpression, if necessary.)

[2] updateAssignments ( ) : AssignedSource [*]

The assignments after an invocation expression are the same as those after the tuple of the expression.

13.2.24 IsolationExpression

An expression used to evaluate its operand expression in isolation.

Generalizations

- UnaryExpression

Synthesized Properties
None

Derived Properties
None

Constraints
[1] isolationExpressionLowerDerivation

An isolation expression has the multiplicity lower bound of its operand expression.

[2] isolationExpressionTypeDerivation

An isolation expression has the type of its operand expression.

[3] isolationExpressionUpperDerivation

An isolation expression has the multiplicity upper bound of its operand expression.

Helper Operations
None

13.2.25 IsUniqueExpression
A sequence expansion expression with a isUnique.

Generalizations
- SequenceExpansionExpression

Synthesized Properties
None

Derived Properties
None

Constraints
[1] isUniqueExpressionExpressionArgument

The argument of an isUnique expression must have a multiplicity upper bound of 1.

[2] isUniqueExpressionLowerDerivation

An isUnique expression has a multiplicity lower bound of 1.

[3] isUniqueExpressionTypeDerivation

An isUnique expression has the type Boolean.

[4] isUniqueExpressionUpperDerivation

An isUnique expression has a multiplicity upper bound of 1.

Helper Operations
None

13.2.26 LeftHandSide

The left-hand side of an assignment expression.

NOTE: The derivations for the derived properties of LeftHandSide are specific to its various subclasses.

Generalizations

- SyntaxElement

Synthesized Properties

- index : Expression [0..1]

  An optional expression that evaluates to an index into the values of an ordered sequence.

Derived Properties

- assignmentAfter : AssignedSource [*]

  The assignments after the left-hand side.

- assignmentBefore : AssignedSource [*]

- lower : Integer

  The statically determined lower bound of the multiplicity of this left-hand side.

- referent : ElementReference [0..1]

  A reference to the assignable element denoted by this left-hand side, if one exists (i.e., the left-hand side is not for the first assignment of a local name).

- type : ElementReference [0..1]

  A reference to the element that specifies the statically determined type of this left-hand side (if any).

- upper : UnlimitedNatural

  The statically determined upper bound of the multiplicity of this left-hand side.

Constraints

[1] leftHandSideIndexExpression

If a left-hand side has an index, then the index expression must have a multiplicity upper bound no greater than 1.

Helper Operations

None

13.2.27 LinkOperationExpression

An expression used to create or destroy the links of an association.

Generalizations

- InvocationExpression

Synthesized Properties

- associationName : QualifiedName
  The qualified name of the association whose links are being acted on.

- operation : String
  The name of the link operation.

Derived Properties

- isClear : Boolean
  Whether the operation is clearing the association.

- isCreation : Boolean
  Whether the operation is for link creation.

Constraints

[1] linkOperationExpressionArgumentCompatibility
Each argument expression must be assignable to its corresponding expression.

[2] linkOperationExpressionAssociationReference
The qualified name of a link operation expression must resolve to a single association.

[3] linkOperationExpressionFeatureDerivation
There is no feature for a link operation expression.

[4] linkOperationExpressionIsClearDerivation
A link operation expression is for clearing an association if the operation is "clearAssoc".

[5] linkOperationExpressionIsCreationDerivation
A link operation expression is for link creation if its operation is "createLink".

[6] linkOperationExpressionReferentDerivation
The referent for a link operation expression is the named association.

Helper Operations

[1] parameterElements ( ) : ElementReference [*]
For a clear association operation, returns a single, typeless parameter. Otherwise, returns the ends of the named association.

13.2.28 LiteralExpression

An expression that comprises a primitive literal.

Generalizations

- Expression

Synthesized Properties

None

Derived Properties
None

Constraints
[1] literalExpressionLowerDerivation

The multiplicity lower bound of a literal expression is always 1.

[2] literalExpressionUpperDerivation

The multiplicity upper bound of a literal expression is always 1.

Helper Operations
None

13.2.29 LogicalExpression
A binary expression with a logical operator.

Generalizations
- BinaryExpression

Synthesized Properties
None

Derived Properties
- isBitStringConversion1 : Boolean
  Whether the first operand expression requires BitString conversion.

- isBitStringConversion2 : Boolean
  Whether the second operand expression requires BitString conversion.

- isBitWise : Boolean
  Whether this is a bit-wise logical operation on bit strings.

Constraints
[1] logicalExpressionIsBitStringConversion1Derivation

BitString conversion is required if the first operand expression of a logical expression has a type that conforms to type Integer.

[2] logicalExpressionIsBitStringConversion2Derivation

BitString conversion is required if the second operand expression of a logical expression has a type that conforms to type Integer.

[3] logicalExpressionIsBitWiseDerivation

A logical expression is bit-wise if the type of its first operand is not Boolean.

[4] logicalExpressionLowerDerivation

A logical expression has a multiplicity lower bound of 01.

[5] logicalExpressionOperands

The operands of a logical expression must have types that conforms to type Boolean, Integer or BitString. However, if one of the operands is Boolean, then the other must also be Boolean.

[6] logicalExpressionTypeDerivation

A logical expression has type Boolean if it is not bit-wise and type BitString if it is bit-wise.

[7] logicalExpressionUpperDerivation

A logical expression has a multiplicity upper bound of 1.

Helper Operations

None

13.2.30 NameBinding

An unqualified name, optionally with a template binding.

Generalizations

- SyntaxElement

Synthesized Properties

- binding : TemplateBinding [0..1]

  The template binding to be used, if the name denotes a template.

- name : String

  An unqualified name. For names that appeared as unrestricted names in the input text, the string value here excludes the surrounding single quote characters and has any escape sequences resolved to the denoted special characters.

Derived Properties

None

Constraints

None

Helper Operations

None

13.2.31 NamedExpression

A pairing of a parameter name and an argument expression in a tuple.

Generalizations

- SyntaxElement

Synthesized Properties

- expression : Expression

  The argument expression.

- index : Expression [0..1]

  An expression whose value gives an index into an ordered parameter. (This is only used in link operation expressions.)

- isRealConversion : Boolean

  Whether the argument expression requires Real conversion.

- name : String

  The parameter name.

Derived Properties

- isBitStringConversion : Boolean

  Whether the argument expression requires bit string conversion.

- isCollectionConversion : Boolean

  Whether the argument expression requires collection conversion.

Constraints

[1] namedExpressionIsBitStringConversionDerivation

Bit string conversion is required if the type of the type of the corresponding parameter is BitString, or a collection class whose sequence type is BitString, and the type of the argument expression is not BitString.

[2] namedExpressionIsCollectionConversionDerivation

Collection conversion is required if the type of the corresponding parameter is a collection class and the type of the argument expression is not.

[3] namedExpressionIsRealConversionDerivation

Real conversion is required if the type of the corresponding parameter is a type that conforms to type Real, or a collection class whose sequence type is a type that conforms to type Real, and the type of the argument expression is not a type that conforms to type Real.

Helper Operations
None

13.2.32 NamedTemplateBinding

A template binding in which the arguments are matched to formal template parameters by name.

Generalizations

- TemplateBinding

Synthesized Properties

- substitution : TemplateParameterSubstitution [1..*]

  The substitutions of arguments for template parameters.

Derived Properties

None

Constraints

None

Helper Operations

None

13.2.33 NamedTuple

A tuple in which the arguments are matched to parameters by name.

Generalizations

- Tuple

Synthesized Properties

- namedExpression : NamedExpression [*]

  The argument expressions for this tuple paired with the corresponding parameter names.

Derived Properties

None

Constraints

[1] namedTupleArgumentNames

The name of a named expression of a named tuple must be the name of a parameter of the invocation the tuple is for. No two named expressions may have the same name.

Helper Operations

None

13.2.34 NameExpression

An expression that comprises a name reference.

Generalizations

- Expression

Synthesized Properties

- name : QualifiedName

  The qualified name referenced in this expression. (For a local name, this will actually have no qualification.)

Derived Properties

- assignment : AssignedSource [0..1]

  The assigned source for the referenced name, if the name is a local or parameter name.

- enumerationLiteral : ElementReference [0..1]
  The identified enumeration literal, if the referenced name is for an enumeration literal.

- propertyAccess : PropertyAccessExpression [0..1]
  The equivalent property access expression, if the referenced name disambiguates to a feature reference.

Constraints

[1] nameExpressionAssignmentDerivation

If the name in a name expression is a local name or parameter name for an `out` parameter, then its assignment is its assigned source before the expression. If the name is a parameter name for an `in` or `inout` parameter, then its assignment is its assigned source before the expression, if it has one, and, otherwise, it is a new assignment whose source is the named parameter and whose type and multiplicity are those of the parameter.

[2] nameExpressionEnumerationLiteralDerivation

If the name in a name expression resolves to an enumeration literal name, then that is the enumeration literal for the expression.

[3] nameExpressionLowerDerivation

The multiplicity lower bound of a name expression is determined by its name.

[4] nameExpressionPropertyAccessDerivation

If the name in a name expression disambiguates to a feature reference, then the equivalent property access expression has the disambiguation of the name as its feature. The assignments before the property access expression are the same as those before the name expression.

[5] nameExpressionResolution

If the name referenced by this expression is not a disambiguated feature reference or a local or parameter name, then it must resolve to exactly one enumeration literal.

[6] nameExpressionTypeDerivation

The type of a name expression is determined by its name. If the name is a local name or parameter with an assignment, then the type of the name expression is the best known type of that assignment. If the name is an enumeration literal, then the type of the name expression is the corresponding enumeration. If the name disambiguates to a feature reference, then the type of the name expression is the type of the equivalent property access expression.

[7] nameExpressionUpperDerivation

The multiplicity upper bound of a name expression is determined by its name.

Helper Operations

[1] adjustMultiplicity(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the name does not disambiguate to a feature reference, then it is considered known null if the condition is true and known non-null if the condition is false.

[2] adjustType(assignments : AssignedSource [*], subtype : ElementReference) : AssignedSource [*]

If the name does not disambiguate to a feature reference, then it is considered to have the given subtype.

[3] declaredType( ) : ElementReference

If a name expression has a derived assignment, then its declared type is the type of that assignment. Otherwise it is the same as the type of the expression.

[4] updateAssignments ( ) : AssignedSource [*]

If propertyAccess is not empty (i.e., if the referenced name disambiguates to a feature reference), then return the assignments after the propertyAccess expression. Otherwise, return the result of the superclass updateAssignments operation.

13.2.35 NameLeftHandSide

A left-hand side that is a name.

Generalizations

- LeftHandSide

Synthesized Properties

- target : QualifiedName

  The name that is the target of the assignment.

Derived Properties

None

Constraints

[1] nameLeftHandSideAssignmentAfterDerivation

If a name left-hand side has an index, then the assignments after the left-hand side are the same as the assignments after the index. If the left-hand side has no index, but its target disambiguates to a feature reference, then the assignments after the left-hand side are the assignments after the feature expression. Otherwise the assignments after the left-hand side are the same as the assignments before the left-hand side.

[2] nameLeftHandSideAssignmentsBefore

If the target of a name left-hand side disambiguates to a feature reference, then the assignments before the expression of the feature reference are the assignments before the left-hand side. If a name left-hand side has an index, then the target must either disambiguate to a feature reference or already have an assigned source, and the assignments before the index expression are the assignments before the left-hand side or, if the target disambiguates to a feature reference, the assignments after the expression of the feature reference.

[3] nameLeftHandSideFeatureExpression
If the target of a name left-hand side disambiguates to a feature reference, then the expression of the feature reference must have a multiplicity upper bound of 1.

[4] nameLeftHandSideIndexedFeature

If the target of a name left-hand side disambiguates to a feature reference, and the left-hand side has an index, then the referent of the feature reference must be ordered and non-unique.

[5] nameLeftHandSideLowerDerivation

If a name left-hand side is indexed, then its lower bound is 0. Otherwise, if the left-hand side is for a local name with an assignment, than its lower bound is that of the assignment, else, if it has a referent, then its lower bound is that of the referent.

[6] nameLeftHandSideNontemplateTarget

The target of a name left-hand side must not have a template binding.

[7] nameLeftHandSideReferentDerivation

If the target of a name left-hand side disambiguates to a structural feature, then the referent of the left-hand side is that feature. If the target resolves to a parameter, then the referent is that parameter. If the target resolves to a local name, then the referent is the assigned source for that local name, if it has one.

[8] nameLeftHandSideTargetAssignment

The target of a name left hand side may not already have an assigned source that is a loop variable definition, an annotation, a sequence expansion expression or a parameter that is an in parameter.

[9] nameLeftHandSideTargetResolution

If the target of a name left-hand side is qualified, then, if it does not disambiguate to a feature, it must have a referent that is a parameter of an operation or behavior that is the current scope the left-hand is in, and, if it does disambiguate to a feature, it must have a single referent that is a structural feature.

[10] nameLeftHandSideTypeDerivation

If a name left-hand side is for a local name with an assignment, then its type is that of that assignment. Otherwise, if the left-hand side has a referent, then its type is the type of that referent.

[11] nameLeftHandSideUpperDerivation

If a name left-hand side is indexed, then its upper bound is 1. Otherwise, if the left-hand side is for a local name with an assignment, than its upper bound is that of the assignment, else, if it has a referent, then its upper bound is that of the referent.

Helper Operations

None

13.2.36 NaturalLiteralExpression

An expression that comprises a natural literal.

Generalizations

- LiteralExpression

Synthesized Properties

- image : String

  The textual image of the literal token for this expression.

Derived Properties

None

Constraints

[1] naturalLiteralExpressionTypeDerivation

The type of a natural literal is the Alf library type Natural.

NOTE: If the context of a natural literal expression unambiguously requires either an Integer or an UnlimitedNatural value, then the result of the literal expression is implicitly downcast to the required type. If the context is ambiguous, however, than an explicit cast to Integer or UnlimitedNatural must be used.

Helper Operations

None

13.2.37 NullCoalescingExpression

An expression that evaluates to the result of its first operand expression, unless that result is null and it has a second operand expression, in which case it evaluates to the result of its second operand expression.

Generalizations

- BinaryExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] nullCoalescingExpressionAssignmentsBefore

The assignments before the first operand expression of a null-coalescing expression are the same as those before the null-coalescing expression. The assignments before the second operand expression are the same as those after the first operand expression.

[2] nullCoalescingExpressionLowerDerivation

The multiplicity lower bound of a null-coalescing expression is the multiplicity lower bound of its first greater than 0; otherwise it is 1, if the multiplicity lower bound of its second operand expression is greater than 0; otherwise, it is 0.

[3] nullCoalescingExpressionTypeDerivation

If one of the operand expressions of a null-coalescing expression is identically null (untyped with multiplicity 0..0), then the type of the null-coalescing expression is the same as the type of the other operand expressions. Otherwise, the type of a null-coalescing expression is the effective common ancestor of the types of its operands, if one exists, and empty, if it does not.

[4] nullCoalescingExpressionUpperDerivation

The multiplicity upper bound of a null-coalescing expression is the maximum of the multiplicity upper bounds of its operands.

Helper Operations

[1] minLowerBound( ) : Integer

The minimum lower bound of an operand of a null-coalescing expression is 0.

[2] maxUpperBound( ) : UnlimitedNatural

The maximum upper bound of an operand of a null-coalescing expression is * (unbounded).

[3] updateAssignments( ) : AssignedSource [*]

If a name has the same assigned source after the second operand expression of a null-coalescing expression as before it, that is its assigned source after the null-coalescing expression. Otherwise, its assigned source after the null-coalescing expression is the null-coalescing expression. If a name is unassigned before the second operand expression but assigned after it, then it has a multiplicity lower bound of 0 after the null-coalescing expression.

13.2.38 NumericUnaryExpression

A unary expression with a numeric operator.

Generalizations

- UnaryExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] numericUnaryExpressionLowerDerivation

A numeric unary expression has the same multiplicity lower bound as its operand expression.

[2] numericUnaryExpressionOperand

The operand expression must have a type that conforms to type Integer or Real and a multiplicity upper bound of 1.

[3] numericUnaryExpressionTypeDerivation

If the operand of a numeric unary expression is of a type that conforms to type Integer, then the expression is Integer. Of the operand is of a type that conforms to type Real, then the type of the expression is Real. Otherwise it has no type.

[4] numericUnaryExpressionUpperDerivation

A numeric unary expression has a multiplicity upper bound of 1.

Helper Operations

None

13.2.39 OutputNamedExpression

A named argument expression for an output parameter.

Generalizations

- NamedExpression

Synthesized Properties

None

Derived Properties

- leftHandSide : LeftHandSide

  The argument expression considered as an assignment left-hand side.

Constraints

[1] outputNamedExpressionForm

The argument for an output parameter must be either be null, a name expression, a property access expression, or a sequence access expression whose primary expression is a name expression or a property access expression.

[2] outputNamedExpressionLeftHandSideDerivation

The equivalent left-hand side for an output named expression depends on the kind of expression. If the expression is a name expression with no disambiguation, then the left-hand side is a name left-hand side with the name from the name expression. If the expression is a name expression that disambiguates to a feature reference, then the left-hand side is a feature left-hand side for that feature reference. If the expression is a property access expression, then the left-hand side is a feature left-hand side for the feature reference of the property access expression. If the expression is a sequence access expression, then the left-hand side is a name left-hand side or feature left-hand side, as above, depending on whether the primary expression of the sequence access expression is a name expression or property access expression, and an index given by the index expression of the sequence access expression. Otherwise the left-hand side is empty.

Helper Operations

None

13.2.40 PositionalTemplateBinding

A template binding in which the arguments are matched to formal template parameters in order by position.

Generalizations

- TemplateBinding

Synthesized Properties

- argumentName : QualifiedName [1..*]

  The arguments for this template binding, to be matched by position to the template parameters.

Derived Properties

None

Constraints

None

Helper Operations

None

13.2.41 PositionalTuple

A tuple in which the arguments are matched to parameters in order by position.

Generalizations

- Tuple

Synthesized Properties

- expression : Expression [*]

  The argument expressions for this tuple, to be matched by position to the invocation parameters.

Derived Properties

None

Constraints

[1] positionalTupleArguments

A positional tuple must not have more arguments than the invocation it is for has parameters.

Helper Operations

None

13.2.42 PropertyAccessExpression

An expression comprising a reference to a structural feature.

Generalizations

- Expression

Synthesized Properties

- featureReference : FeatureReference

  A reference to a structural feature.

Derived Properties

- feature : ElementReference

  The referenced structural feature.

Constraints

[1] propertyAccessExpressionAssignmentsBefore

The assignments before the expression of the feature reference of a property access expression are the same as before the property access expression.

[2] propertyAccessExpressionFeatureDerivation

The feature of a property access expression is the structural feature to which its feature reference resolves.

[3] propertyAccessExpressionFeatureResolution

The feature reference for a property access expression must resolve to a single structural feature.

[4] propertyAccessExpressionLowerDerivation

The multiplicity lower bound of a property access expression is given by the product of the multiplicity lower bounds of the referenced feature and the target expression.

[5] propertyAccessExpressionTypeDerivation

The type of a property access expression is the type of the referenced feature.

[6] propertyAccessExpressionUpperDerivation

The multiplicity upper bound of a property access expression is given by the product of the multiplicity upper bounds of the referenced feature and the target expression.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

The assignments after a property access expression are the same as those after the target expression of its feature reference.

13.2.43 QualifiedName

The representation of a qualified name as a sequence of individual simple names.

Generalizations

- SyntaxElement

Synthesized Properties

- isAmbiguous : Boolean = false

  Whether this qualified name is ambiguous.

- nameBinding : NameBinding [*]
  The sequence of individual name bindings in this qualified name.

Derived Properties

- disambiguation : FeatureReference [0..1]
  The disambiguation into a feature reference of a syntactic element initially parsed as a qualified name.

- isFeatureReference : Boolean
  Indicates whether this qualified name has been disambiguated to a feature reference.

- pathName : String
  The complete path name for the qualified name, with individual name bindings separated by "::" punctuation.

- qualification : QualifiedName [0..1]
  The qualified name corresponding to the qualification part of this qualified name, if any.

- referent : ElementReference [*]
  The possible referents to which this qualified name may resolve. (Note that the UML rules for namespaces in general allow a namespace to contain elements of different kinds with the same name.) If the qualified name is for a template instantiation, then the referent is the equivalent bound element.

- templateName : QualifiedName [0..1]
  If this qualified name is for a template binding, then the name of the template for which this qualified name is a binding.

- unqualifiedName : NameBinding
  The rightmost individual name binding in the qualified name, without the qualification.

Constraints

[1] qualifiedNameDisambiguationDerivation

If a qualified name is not ambiguous or it has a qualification that resolves to a namespace, then it is has no disambiguation. Otherwise, its disambiguation is a feature reference with a name given by the unqualified name of the qualified name and a target expression determined by the disambiguation of the qualification of the qualified name.

[2] qualifiedNameIsFeatureReferenceDerivation

A qualified name is a feature reference is its disambiguation is not empty.

[3] qualifiedNameLocalName

If a qualified name is a local name, then the reference must be within the same local scope as the definition of the named element.

[4] qualifiedNameNonLocalUnqualifiedName

If a qualified name is an unqualified, non-local name, then it must be visible in the current scope of the use of the name.

[5] qualifiedNamePathNameDerivation

The path name for a qualified name consists of the string representation of each of the name bindings, separated by "::" punctuation. The string representation of a name binding is its name followed by the representation of its template binding, if it has one. The string representation of a positional template binding consists of an ordered list of the path names of its argument qualified names separated by commas, all surrounded by the angle brackets "<" and ">". The string representation of a named template binding consists of an ordered list of its template parameter substitutions, each consisting of the formal parameter name followed by "=>" followed by the path name of the argument qualified name, separated by commas, all surrounded by the angle brackets "<" and ">".

[6] qualifiedNameQualificationDerivation

The qualification of a qualified name is a empty if the qualified name has only one name binding. Otherwise it is the qualified name consisting of all the name bindings of the original qualified name except for the last one. The qualification of a qualified name is considered ambiguous if the qualified name is ambiguous and has more than two name bindings.

[7] qualifiedNameQualifiedResolution

If a qualified name has a qualification, then its unqualified name must name an element of the namespace named by the qualification, where the first name in the qualification must name an element of the current scope.

[8] qualifiedNameReferentDerivation

The referents of a qualified name are the elements to which the name may resolve in the current scope, according to the UML rules for namespaces and named elements.

[9] qualifiedNameTemplateBinding

If the unqualified name of a qualified name has a template binding, then the template name must resolve to a template. The template binding must have an argument name for each of the template parameters and each argument name must resolve to a classifier. If the template parameter has constraining classifiers, then the referent of the corresponding argument name must conform to all those constraining classifiers.

[10] qualifiedNameTemplateNameDerivation

If the last name binding in a qualified name has a template binding, then the template name is a qualified name with the same template bindings as the original qualified name, but with the template binding removed on the last name binding.

[11] qualifiedNameUnqualifiedNameDerivation

The unqualified name of a qualified name is the last name binding.

Helper Operations
None

13.2.44 RealLiteralExpression
An expression that comprises a real literal.

Generalizations
- Literal Expressions

Synthesized Properties
- image : String
  The textual image of the literal token for this expression.

Derived Properties
None

Constraints
[1] realLiteralExpressionTypeDerivation
The type of a real literal expression is the Alf library type Real.

Helper Operations
None

13.2.45 RelationalExpression
A binary expression with a relational operator.

Generalizations
- BinaryExpression

Synthesized Properties
None

Derived Properties
- isReal : Boolean
  Whether this is a Real comparison.

- isRealConversion1 : Boolean
  Whether Real conversion is required on the first operand of this expression.

- isRealConversion2 : Boolean
  Whether Real conversion is required on the second operand of this expression.

- isUnlimitedNatural : Boolean
  Whether this is an UnlimitedNatural comparison.

Constraints

[1] relationExpressionIsRealConversion1Derivation

A relational expression requires Real conversion if it is a Real comparison and its first operand is of a type that conforms to type Integer.

[2] relationExpressionIsRealConversion2Derivation

A relational expression requires Real conversion if it is a Real comparison and its second operand is of a type that conforms to type Integer.

[3] relationExpressionIsRealDerivation

A relational expression is a Real comparison if either one of its operations has a type that conforms to type Real.

[4] relationalExpressionIsUnlimitedNaturalDerivation

A relational expression is an UnlimitedNatural comparison if either one of its operands has type UnlimitedNatural.

[5] relationalExpressionLowerDerivation

A relational expression has a multiplicity lower bound of 0 if the lower bound if either operand expression is 0 and 1 otherwise.

[6] relationalExpressionOperandTypes

The operand expressions for a comparison operator must both be of a type that conforms to type Natural, Integer or Real, or both be of types that conform to type Natural or UnlimitedNatural.

[7] relationalExpressionTypeDerivation

The type of a relational expression is Boolean.

[8] relationalExpressionUpperDerivation

A relational expression has a multiplicity upper bound of 1.

Helper Operations

[1] minLowerBound ( ) : Integer

The minimum lower bound is 0 for operands of relational expressions (this allows for the propagation of a null returned from an arithmetic expression used as an operand).

13.2.46 SelectOrRejectExpression

A sequence expansion expression with a select or reject operation.

Generalizations

- SequenceExpansionExpression

Synthesized Properties

None

Derived Properties
None

Constraints
[1] selectOrRejectExpressionArgument

The argument of a select or reject expression must have a type that conforms to type Boolean and a multiplicity upper bound of 1.

[2] selectOrRejectExpressionLowerDerivation

A select or reject expression has a multiplicity lower bound of 0.

[3] selectOrRejectExpressionTypeDerivation

A select or reject expression has the same type as its primary expression.

[4] selectOrRejectExpressionUpperDerivation

A select or reject expression has the same multiplicity upper bound as its primary expression.

Helper Operations
None

13.2.47 SequenceAccessExpression
An expression used to access a specific element of a sequence.

Generalizations
- Expression

Synthesized Properties
- index : Expression

  The expression whose value is the index of the element being accessed in the sequence.

- primary : Expression

  The expression whose value is the sequence being accessed.

Derived Properties
None

Constraints
[1] sequenceAccessExpressionIndexMultiplicity

The multiplicity upper bound of the index of a sequence access expression must be 1.

[2] sequenceAccessExpressionIndexType

The type of the index of a sequence access expression must be Integer.

[3] sequenceAccessExpressionLowerDerivation

The multiplicity lower bound of a sequence access expression is 0.

[4] sequenceAccessExpressionTypeDerivation

The type of a sequence access expression is the same as the type of its primary expression.

[5] sequenceAccessExpressionUpperDerivation

The multiplicity upper bound of a sequence access expression is 1.

Helper Operations

None

13.2.48 SequenceConstructionExpression

An expression used to construct a sequence of values.

Generalizations

- Expression

Synthesized Properties

- elements : SequenceElements [0..1]
  The specification of the elements in the sequence.

- hasMultiplicity : Boolean = false
  Whether the sequence construction expression has a multiplicity indicator.

- typeName : QualifiedName [0..1]
  The name of the type of the elements in the sequence.

Derived Properties

None

Constraints

[1] sequenceConstructionExpressionAssignmentsBefore

If the elements of a sequence construction expression are given by a sequence expression list, then the assignments before the first expression in the list are the same as the assignments before the sequence construction expression, and the assignments before each subsequent expression are the assignments after the previous expression. If the elements are given by a sequence range, the assignments before both expressions in the range are the same as the assignments before the sequence construction expression.

[2] sequenceConstructionExpressionElementType

If the elements of a sequence construction expression are given by a sequence range, then the expression must have a type that conforms to type Integer. If the elements are given by a sequence element list, and the sequence construction expression has a non-empty type, then each expression in the list must have a type that either conforms to the type of the sequence construction expression or is convertible to it by bit string conversion or real conversion.

[3] sequenceConstructionExpressionLowerDerivation

If a sequence construction expression has multiplicity, then its multiplicity lower bound is given by its elements, if this is not empty, and zero otherwise. If a sequence construction expression does not have

multiplicity, then its multiplicity lower bound is one.

[4] sequenceConstructionExpressionType

If the type name of a sequence construction expression is not empty, then it must resolve to a non-template classifier. If the expression does not have multiplicity, then its type must be a collection class.

[5] sequenceConstructionExpressionTypeDerivation

If the type name of a sequence construction expression is not empty, then the type of the expression is the classifier to which the type name resolves.

[6] sequenceConstructionExpressionUpperDerivation

If a sequence construction expression has multiplicity, then its multiplicity upper bound is given by its elements, if this is not empty, and zero otherwise. If a sequence construction expression does not have multiplicity, then its multiplicity upper bound is one.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

If the elements of the sequence construction expression are given by a sequence expression list, then return the assignments after the last expression in the list (if the list is empty, then return the assignments before the sequence construction expression). If the elements are given by a sequence range, then return the union of the assignments after each of the expressions in the range.

13.2.49 SequenceElements

A specification of the elements of a sequence.

Generalizations

- SyntaxElement

Synthesized Properties

None

Derived Properties

- lower : Integer

  The multiplicity lower bound of the elements of the sequence. The derivation for this property is given in the subclasses of SequenceElements.

- upper : UnlimitedNatural

  The multiplicity upper bound of the elements of the sequence. The derivation for this property is given in the subclasses of SequenceElements.

Constraints

None

Helper Operations

None

13.2.50 SequenceExpansionExpression

An expression used to carry out one of a predefined set of operations over each of the elements in a sequence.

Generalizations

- Expression

Synthesized Properties

- argument : Expression

  The argument expression. The exact form required for this expression depends on which expansion operation is being carried out.

- operation : String

  The name of the operation to be carried out.

- primary : ExtentOrExpression

  The class name or primary expression that evaluates to the sequence to be acted on.

- variable : String

  The name of the expansion variable available as a local name within the argument expression.

Derived Properties

- variableSource : AssignedSource

  The assigned source for the expansion variable within the argument expression. The source is actually the sequence expansion expression itself.

Constraints

[1] sequenceExpansionExpressionAssignmentsAfterArgument

The assignments after the argument expression of a sequence expansion expression must be the same as the assignments before the argument expression.

[2] sequenceExpansionExpressionAssignmentsBeforeArgument

The assignments before the argument expression of a sequence expansion expression include those after the primary expression plus one for the expansion variable.

[3] sequenceExpansionExpressionAssignmentsBeforePrimary

The assignments before the primary expression of a sequence expansion expression are the same as the assignments before the sequence expansion expression.

[4] sequenceExpansionExpressionVariableName

The expansion variable name may not conflict with any name already assigned after the primary expression.

[5] sequenceExpansionExpressionVariableSourceDerivation

The assigned source for the expansion variable of a sequence expansion expression is the expression itself. The type for the assignment is the type of the primary expression of the sequence expansion expression and the multiplicity lower and upper bounds are 1.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

The assignments after a sequence expansion expression are the same as after its primary expression.

13.2.51 SequenceExpressionList

A specification of the elements of a sequence using a list of expressions.

Generalizations

- SequenceElements

Synthesized Properties

- element : Expression [*]

  The list of expressions whose values determine the elements of the sequence.

Derived Properties

None

Constraints

[1] sequenceExpressionListLowerDerivation

The multiplicity lower bound of the elements of a sequence expression list is given by the sum of the lower bounds of each of the expressions in the list.

[2] sequenceExpressionListUpperDerivation

The multiplicity upper bound of the elements of a sequence expression list is given by the sum of the upper bounds of each of the expressions in the list. If any of the expressions in the list have an unbounded upper bound, then the sequence expression list also has an unbounded upper bound.

Helper Operations

None

13.2.52 SequenceOperationExpression

An expression used to invoke a behavior as if it was an operation on a target sequence as a whole.

Generalizations

- InvocationExpression

Synthesized Properties

- operation : QualifiedName [0..1]

  The qualified name of the behavior being invoked.

- primary : ExtentOrExpression
  The expression or class name whose value is gives the sequence to be operated on.

Derived Properties

- isBitStringConversion : Boolean
  Whether type primary expression requires BitString conversion.

- isCollectionConversion : Boolean
  Whether the primary expression requires collection conversion.

- leftHandSide : LeftHandSide [0..1]
  The effective left-hand side corresponding to the primary expression, if the sequence operation is "in place" (that is, has a first parameter with direction inout).

Constraints

[1] sequenceOperationExpressionArgumentCompatibility

The type of an input argument expression of a sequence operation parameter must be assignable to its corresponding parameter. The type of an output parameter must be assignable to its corresponding argument expression. (Note that this implies that the type of an argument expression for an inout parameter must be the same as the type of that parameter.)

[2] sequenceOperationExpressionAssignmentsAfter

A local name that is assigned in the primary expression of a sequence operation expression may not be assigned in any expression in the tuple of the sequence operation expression.

[3] sequenceOperationExpressionAssignmentsBefore

The assignments before the primary expression of a sequence operation expression are the same as the assignments before the sequence operation expression.

[4] sequenceOperationExpressionFeatureDerivation

There is no feature for a sequence operation expression.

[5] sequenceOperationExpressionIsBitStringConversionDerivation

BitString conversion is required if type of the first parameter of the referent of a sequence operation expression is BitString and either the type of its primary expression is Integer or collection conversion is required and the type of its primary expression is a collection class whose sequence type is Integer.

[6] sequenceOperationExpressionIsCollectionConversionDerivation

Collection conversion is required if the type of the primary expression of a sequence operation expression is a collection class and the multiplicity upper bound of the primary expression is 1.

[7] sequenceOperationExpressionLeftHandSideDerivation

If the operation of a sequence operation expression has a first parameter whose direction is inout, then the effective left-hand side for the expression is constructed as follows: If the primary is a name expression, then the left-hand side is a name left-hand side with the name from the name expression as

its target. If the primary is a property access expression, then the left-hand side is a feature left hand side with the feature reference from the property access expression as its feature. If the primary is a sequence access expression whose primary is a name expression or a property access expression, then the left-hand side is constructed from the primary of the sequence access expression as given previously and the index of the sequence access expression becomes the index of the left-hand side.

[8] sequenceOperationExpressionOperationReferent

There must be a single behavior that is a resolution of the operation qualified name of a sequence operation expression with a least one parameter, whose first parameter has direction in or inout, has multiplicity [0..*] and to which the target primary expression is assignable. If any resolution of the operation name is a template behavior, then the implicit template binding of that behavior, if legal, is used to check the assignability of the target primary expression.

[9] sequenceOperationExpressionReferentDerivation

The referent for a sequence operation expression is the behavior named by the operation for the expression.

[10] sequenceOperationExpressionTargetCompatibility

If the first parameter of the referent has direction inout, then the parameter type must have the same type as the primary expression, the primary expression must have the form of a left-hand side and, if the equivalent left-hand side is for a local name, that name must already exist. The first parameter must be assignable to the effective left-hand side.

Helper Operations

[1] adjustAssignments(assignments : AssignedSource [*], condition : Boolean) : AssignedSource [*]

If the invoked behavior is CollectionFunctions::isEmpty or SequenceFunctions::IsEmpty, then check the primary expression for known nulls and non-nulls using the given truth condition. If the invoked behavior is CollectionFunctions::notEmpty or SequenceFunctions::NotEmpty, then check the primary expression for known nulls and non-nulls using the negation of the given truth condition.

[2] parameterElements ( ) : ElementReference [*]

Returns the list of parameter elements from the superclass operation, with the first parameter removed (since the argument for the first parameter is given by the primary expression of a sequence operation expression, not in its tuple).

[3] updateAssignments ( ) : AssignedSource [*]

The assignments after a sequence operation expression include those made in the primary expression and those made in the tuple and, for an "in place" operation (one whose first parameter is inout), that made by the sequence operation expression itself.

13.2.53 SequenceRange

A specification of the elements of a sequence as a range of integers.

Generalizations

- SequenceElements

Synthesized Properties

- rangeLower : Expression

  The expression whose value gives the lower bound for the range.

- rangeUpper : Expression

  The expression whose value gives the upper bound for the range.

Derived Properties

None

Constraints

[1] sequenceRangeAssignments

A local name may be defined or reassigned in at most one of the expressions of a sequence range.

[2] sequenceRangeExpressionMultiplicity

Both expression in a sequence range must have a multiplicity upper bound of 1.

[3] sequenceRangeLowerDerivation

The multiplicity lower bound of a sequence range is 0.

[4] sequenceRangeUpperDerivation

The multiplicity uper bound of a sequence range is * (since it is not possible, in general, to statically determine a more constrained upper bound).

Helper Operations

None

13.2.54 SequenceReductionExpression

An expression used to reduce a sequence of values effectively by inserting a binary operation between the values.

Generalizations

- Expression

Synthesized Properties

- behaviorName : QualifiedName

  The name of the behavior to be used as the reducer.

- isOrdered : Boolean = false

  Whether this is an ordered reduction or not.

- primary : ExtentOrExpression

  The target class name or primary expression for the reduction.

Derived Properties

- referent : ElementReference

A reference to the behavior to be used as the reducer.

Constraints

[1] sequenceReductionExpressionAssignmentsBefore

The assignments before the target expression of a sequence reduction expression are the same as the assignments before the sequence reduction expression.

[2] sequenceReductionExpressionBehavior

The behavior name in a sequence reduction expression must denote a behavior.

[3] sequenceReductionExpressionBehaviorParameters

The referent behavior must have two in parameters, a return parameter and no other parameters. The parameters must all have the same type as the argument expression and multiplicity [1..1].

[4] sequenceReductionExpressionLowerDerivation

A sequence reduction expression has a multiplicity lower bound of 1.

[5] sequenceReductionExpressionReferentDerivation

The referent for a sequence reduction expression is the behavior denoted by the behavior name of the expression.

[6] sequenceReductionExpressionTypeDerivation

A sequence reduction expression has the same type as its primary expression.

[7] sequenceReductionExpressionUpperDerivation

A sequence reduction expression has a multiplicity upper bound of 1.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

The assignments after a sequence reduction expression are the same as after its primary expression.

13.2.55 ShiftExpression

Generalizations

- BinaryExpression

Synthesized Properties

None

Derived Properties

- isBitStringConversion : Boolean

  Whether the first operand expression requires BitString conversion.

Constraints

[1] shiftExpressionIsBitStringConversionDerivation

BitString conversion is required if the first operand expression of a shift expression has a type that conforms to type Integer.

[2] shiftExpressionLowerDerivation

A shift expression has a multiplicity lower bound of 1.

[3] shiftExpressionOperands

The first operand expression of a shift expression must have a type that conforms to the type BitString or Integer. The second operand expression must have a type that conforms to the type Integer.

[4] shiftExpressionTypeDerivation

A shift expression has type BitString.

[5] shiftExpressionUpperDerivation

A shift expression has a multiplicity upper bound of 1.

Helper Operations

None

13.2.56 StringLiteralExpression

An expression that comprises a String literal.

Generalizations

- LiteralExpression

Synthesized Properties

- image : String

  The textual image of the literal token for this expression, with quote characters removed and escaped sequences resolved to the denoted special character.

Derived Properties

None

Constraints

[1] stringLiteralExpressionTypeDerivation

The type of a string literal expression is String.

Helper Operations

None

13.2.57 SuperInvocationExpression

An invocation expression used to invoke an operation of a superclass.

Generalizations

- InvocationExpression

Synthesized Properties

- target : QualifiedName [0..1]

  The name of the operation to be invoked, optionally qualified with the name of the appropriate superclass.

Derived Properties

None

Constraints

[1] superInvocationExpressionConstructorCall

If the referent is the method of a constructor operation, the super invocation expression must occur in an expression statement at the start of the definition for the method of a constructor operation, such that any statements preceding it are also super constructor invocations.

[2] superInvocationExpressionDestructorCall

If the referent is the method of a destructor operation, the super invocation expression must occur in an within the method of a destructor operation.

[3] superInvocationExpressionFeatureDerivation

There is no feature for a super invocation.

[4] superInvocationExpressionImplicitTarget

If the target is empty, the super invocation expression must occur within the method of an operation of a class with a single superclass and the referent must be the method of a constructor operation of that superclass.

[5] superInvocationExpressionOperation

It must be possible to identify a single valid operation denoted by the target of a super invocation expression that satisfies the overloading resolution rules.

[6] superInvocationExpressionQualification

If the target has a qualification, then this must resolve to one of the superclasses of the current context class.

[7] superInvocationExpressionReferentDerivation

The referent of a super invocation expression is the method behavior of the operation identified using the overloading resolution rules.

Helper Operations

None

13.2.58 TemplateBinding

A list of type names used to provide arguments for the parameters of a template.

Generalizations

- SyntaxElement

Synthesized Properties

None

Derived Properties

None

Constraints

None

Helper Operations

None

13.2.59 TemplateParameterSubstitution

A specification of the substitution of an argument type name for a template parameter.

Generalizations

- SyntaxElement

Synthesized Properties

- argumentName : QualifiedName

  The name of the argument type.

- parameterName : String

  The name of the template parameter.

Derived Properties

None

Constraints

None

Helper Operations

None

13.2.60 ThisExpression

An expression comprising the keyword “this”.

Generalizations

- Expression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] thisExpressionLowerDerivation

The multiplicity lower bound of a this expression is always 1.

[2] thisExpressionTypeDerivation

The static type of a this expression is the statically determined context classifier for the context in which the this expression occurs.

[3] thisExpressionUpperDerivation

The multiplicity upper bound of a this expression is always 1.

Helper Operations

None

13.2.61 Tuple

A list of expressions used to provide the arguments for an invocation.

Generalizations

- SyntaxElement

Synthesized Properties

- invocation : InvocationExpression

  The invocation expression of which this tuple is a part.

Derived Properties

- input : NamedExpression [*]

  The argument expressions from this tuple, matched to the input parameters (direction in and inout) of the invocation. An empty sequence construction expression is included for any input parameter that is not explicitly matched in the tuple.

- output : OutputNamedExpression [*]

  The argument expressions from this tuple, matched to the output parameters (direction inout and out) of the invocation. An empty sequence construction expression is included for any output parameter that is not explicitly matched in the tuple.

Constraints

[1] tupleAssignmentsAfter

A name may be assigned in at most one argument expression of a tuple.

[2] tupleAssignmentsBefore

The assignments before each expression in a tuple are the same as the assignments before the tuple, except in the case of a name expression that defines a new local name, in which case the assigned source for the new name is included in the assignments before the name expression. (Note that the assigned source for a new name is included before the name expression so that the nameExpressionResolution constraint is not violated.) The assignments before the tuple are the same as the assignments after the feature reference of the invocation of the tuple, if the invocation has one, or otherwise the assignments

before the invocation.

[3] tupleInputDerivation

A tuple has the same number of inputs as its invocation has input parameters. For each input parameter, the tuple has a corresponding input with the same name as the parameter and an expression that is the matching argument from the tuple, or an empty sequence construction expression if there is no matching argument.

[4] tupleNullInputs

An input parameter may only have a null argument if it has a multiplicity lower bound of 0.

[5] tupleOutputDerivation

A tuple has the same number of outputs as its invocation has output parameters. For each output parameter, the tuple has a corresponding output with the same name as the parameter and an expression that is the matching argument from the tuple, or an empty sequence construction expression if there is no matching argument.

[6] tupleOutputs

An output parameter may only have a null argument if it is an out parameter.

Helper Operations

None

13.2.62 UnaryExpression

An expression consisting of an operator acting on a single operand expression.

Generalizations

- Expression

Synthesized Properties

- operand : Expression

  The expression giving the operand.

- operator : String

  The symbol representing the operator.

Derived Properties

None

Constraints

[1] unaryExpressionAssignmentsBefore

The assignments before the operand of a unary expression are the same as those before the unary expression.

Helper Operations

[1] updateAssignments ( ) : AssignedSource [*]

By default, the assignments after a unary expression are the same as those after its operand expression.

13.2.63 UnboundedLiteralExpression

An expression that comprises an unbounded value literal.

Generalizations

- LiteralExpression

Synthesized Properties

None

Derived Properties

None

Constraints

[1] unboundedLiteralExpressionTypeDerivation

The type of an unbounded literal expression is UnlimitedNatural.

Helper Operations

None

14 Statements Abstract Syntax

14.1 Overview

The `Alf::Syntax::Statement` package contains the abstract syntax model for statements. The syntax and semantics of statements are discussed in Clause 9. Their mapping to UML is given in Clause 18.

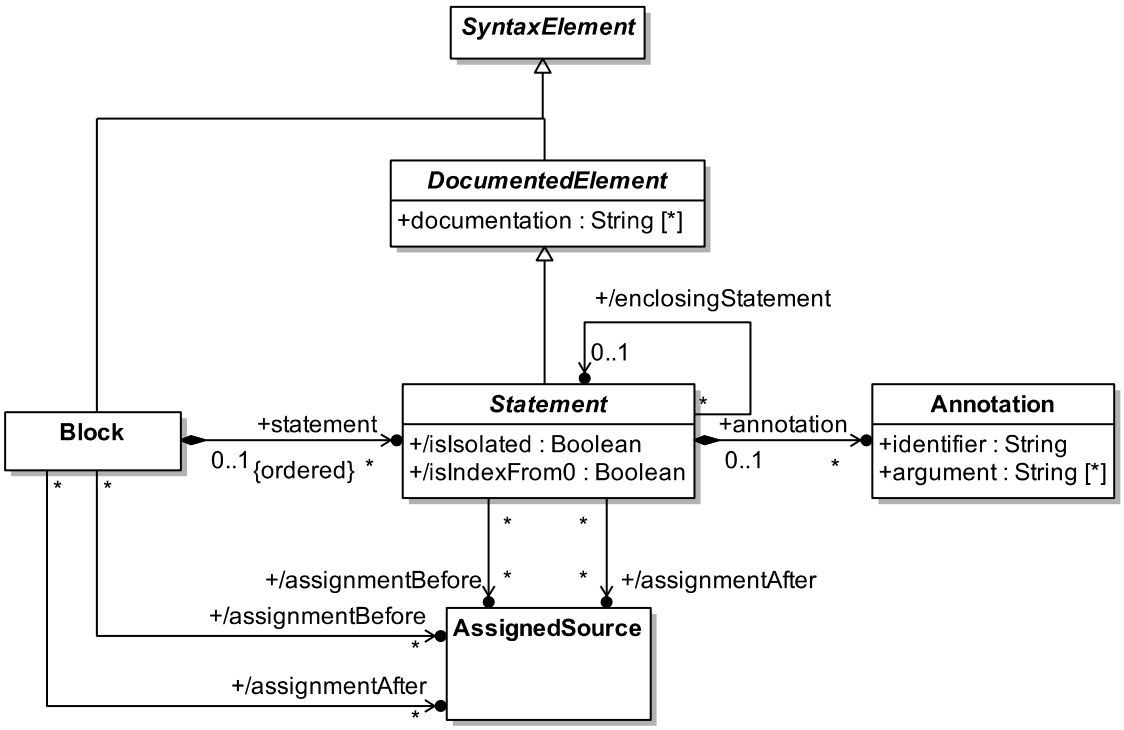

Figure 14.1 Statements and Blocks

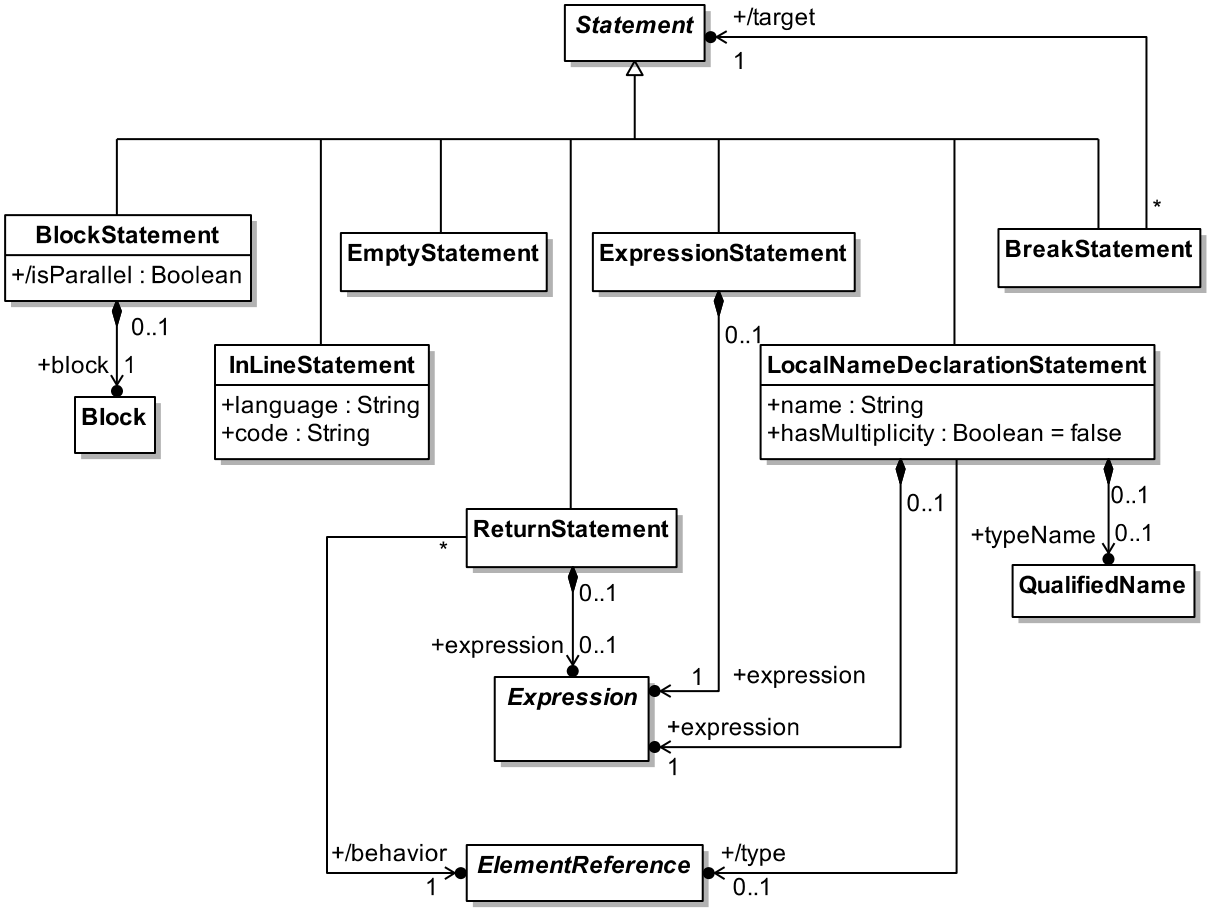

Figure 14.2 Simple Statements

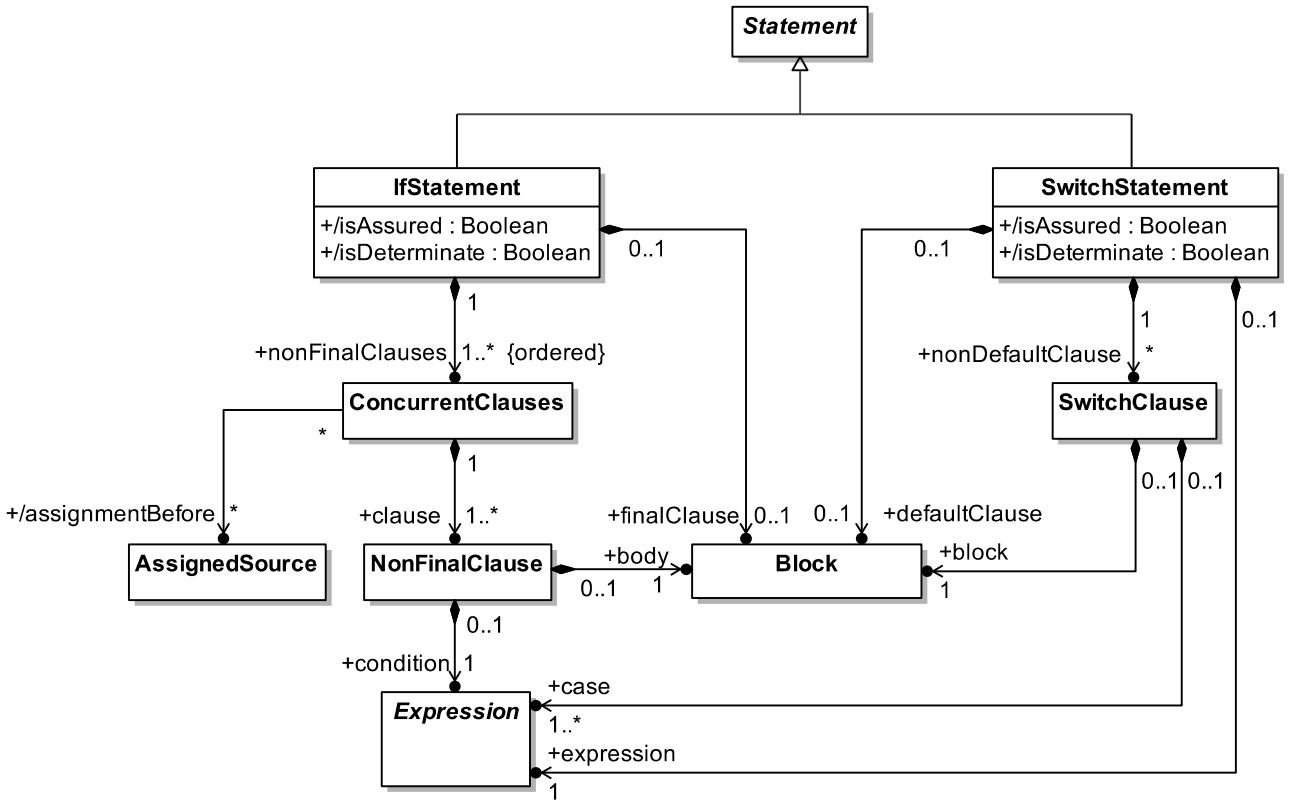

Figure 14.3 Conditional Statements

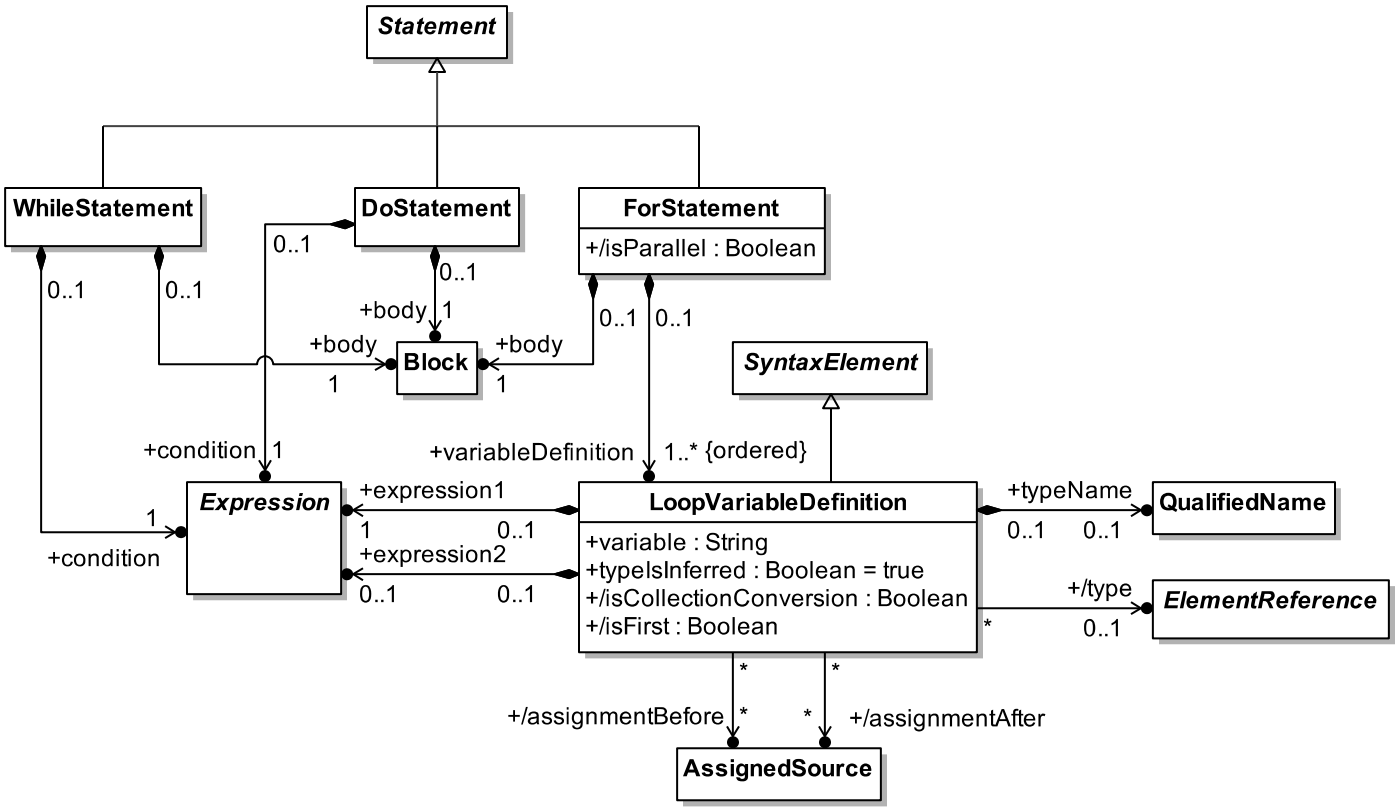

Figure 14.4 Loop Statements

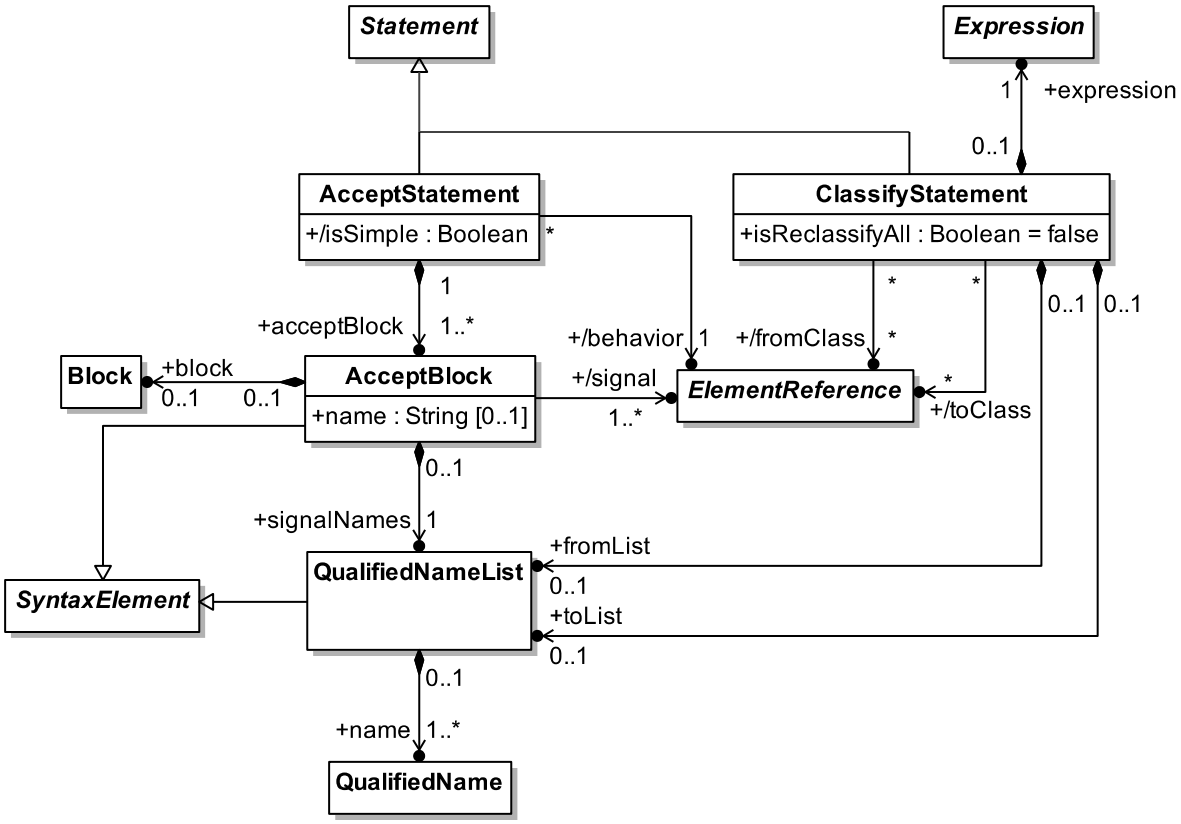

Figure 14.5 `accept` and `classify` Statements

14.2 Class Descriptions

14.2.1 AcceptBlock

A block of an accept statement that accepts one or more signals.

Generalizations

- SyntaxElement

Synthesized Properties

- block : Block [0..1]
  The body of the accept block, executed if one of the named signals is received.

- name : String [0..1]
  The local name to which an accepted signal instance will be assigned.

- signalNames : QualifiedNameList
  A list of names of the signals accepted by this accept block.

Derived Properties

- signal : ElementReference [1..*]
  The signals denoted by the signal names of the accept block.

Constraints

[1] acceptBlockSignalDerivation

The signals of an accept block are the referents of the signal names of the accept block.

[2] acceptBlockSignalNames

All signal names in an accept block must resolve to signals.

Helper Operations

None

14.2.2 AcceptStatement

A statement used to accept the receipt of instances of one or more signals.

Generalizations

- Statement

Synthesized Properties

- acceptBlock : AcceptBlock [1..*]

  One or more blocks for accepting alternate groups of signals.

Derived Properties

- behavior : ElementReference

  The behavior containing the accept statement.

- isSimple : Boolean

  Whether the accept statement is simple or not.

Constraints

[1] acceptStatementAssignmentsAfter

If a name is assigned in any block of an accept statement, then the assigned source of the name after the accept statement is the accept statement itself.

[2] acceptStatementAssignmentsBefore

The assignments before any block of an accept statement are the assignments before the accept statement.

[3] acceptStatementCompoundAcceptLocalName

For a compound accept statement, a local name defined in an accept block has the accept block as its assigned source before the block associated with the accept block. The type of the local name is the effective common ancestor of the specified signals for that accept clause, if one exists, and it is untyped otherwise. However, the local name is considered unassigned after the accept statement.

[4] acceptStatementContext

An accept statement can only be used within the definition of an active behavior or the classifier behavior of an active class.

[5] acceptStatementEnclosedStatements

The enclosing statement for all statements in the blocks of all accept blocks of an accept statement is the accept statement.

[6] acceptStatementIsSimpleDerivation

An accept statement is simple if it has exactly one accept block and that accept block does not have a block.

[7] acceptStatementNames

Any name defined in an accept block of an accept statement must be unassigned before the accept statement.

[8] acceptStatementNewAssignments

Any name that is unassigned before an accept statement and is assigned in one or more blocks of the accept statement, has, after the accept statement, a type that is is the effective common ancestor of the types of the name in each block in which it is defined, with a multiplicity lower bound that is the minimum of the lower bound for the name in each block (where it is considered to have multiplicity lower bound of zero for blocks in which it is not defined), and a multiplicity upper bound that is the maximum for the name in each block in which it is defined.

[9] acceptStatementSignals

The containing behavior of an accept statement must have receptions for all signals from all accept blocks of the accept statement. No signal may be referenced in more than one accept block of an accept statement.

[10] acceptStatementSimpleAcceptLocalName

A local name specified in the accept block of a simple accept statement has the accept statement as its assigned source after the accept statement. The type of the local name is the effective common ancestor of the specified signals, if one exists, and it is untyped otherwise.

Helper Operations

[1] hasReturnValue( ) : Boolean

An accept statement has a return value if all of its accept clauses have return values.

14.2.3 Annotation

An identified modification to the behavior of an annotated statement.

Generalizations

- SyntaxElement

Synthesized Properties

- argument : String [*]

  If permitted by the annotation, an optional list of local names relevant to the annotation.

- identifier : String

The name of the annotation.

Derived Properties
None

Constraints
None

Helper Operations
None

14.2.4 Block

A grouped sequence of statements.

Generalizations

- SyntaxElement

Synthesized Properties

- statement : Statement [*]

  The ordered sequence of statements in the block.

Derived Properties

- assignmentAfter : AssignedSource [*]

  The assigned sources for local names available lexically after this block. This includes not only any assignments made within the block, but also any assignments that are unchanged from before the block.

- assignmentBefore : AssignedSource [*]

  The assigned sources for local names available lexically before this block.

Constraints

[1] blockAssignmentAfterDerivation

If a block is not empty, then the assignments after the block are the same as the assignments after the last statement of the block. Otherwise they are the same as the assignments before the block.

[2] blockAssignmentsBefore

The assignments before the first statement of a block are the same as the assignments before the block.

[3] blockAssignmentsBeforeStatements

The assignments before each statement in a block other than the first are the same as the assignments after the previous statement.

Helper Operations

[1] hasReturnValue( ) : Boolean

A block has a return value if any of its statements has a return value.

14.2.5 BlockStatement

A statement that executes a block.

Generalizations

- Statement

Synthesized Properties

- block : Block

  The block to be executed.

Derived Properties

- isParallel : Boolean

  Whether the statements in the block of this block statement should be executed concurrently.

Constraints

[1] blockStatementAssignmentsAfter

The assignments after a block statement are the same as the assignments after the block of the block statement.

[2] blockStatementAssignmentsBefore

The assignments before the block of a block statement are the same as the assignments before the block statement.

[3] blockStatementEnclosedStatements

The enclosing statement for all the statements in the block of a block statement is the block statement.

[4] blockStatementIsParallelDerivation

A block statement is parallel if it has a @parallel annotation.

[5] blockStatementParallelAssignments

In a parallel block statement, any name assigned in one statement of the block may not be further assigned in any subsequent statement in the same block.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

In addition to an @isolated annotation, a block statement may have a @parallel annotation. It may not have any arguments.

[2] hasReturnValue( ) : Boolean

A block statement has a return value if its block has a return value.

14.2.6 BreakStatement

A statement that causes the termination of execution of an immediately enclosing block.

Generalizations

- Statement

Synthesized Properties

None

Derived Properties

- target : Statement

  The enclosing statement that is terminated by this break statement.

Constraints

[1] breakStatementNonparallelTarget

The target of a break statement may not have a @parallel annotation.

[2] breakStatementTargetDerivation

The target of a break statement is the innermost switch, while, do or for statement enclosing the break statement.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

A break statement may not have any annotations.

14.2.7 ClassifyStatement

A statement that changes the classification of an object.

Generalizations

- Statement

Synthesized Properties

- expression : Expression

  The expression to be evaluated to obtain the object to be reclassified.

- fromList : QualifiedNameList [0..1]

  A list of names of classes to be removed as types of the object.

- isReclassifyAll : Boolean = false

  Whether this classify statement reclassifies all types of the target object.

- toList : QualifiedNameList [0..1]

  A list of names of classes to be added as types of the object.

Derived Properties

- fromClass : ElementReference [*]

  The classes denoted by the names in the from list.

- toClass : ElementReference [*]

  The classes denoted by the names in the to list.

Constraints

[1] classifyStatementAssignmentsAfter

The assignments after a classify statement are the same as the assignments after its expression.

[2] classifyStatementAssignmentsBefore

The assignments before the expression of a classify statement are the same as the assignments before the statement.

[3] classifyStatementClasses

All the from and to classes of a classify statement must be subclasses of the declared type of the target expression and none of them may have a common superclass that is a subclass of the declared type of the target expression (that is, they must be disjoint subclasses).

[4] classifyStatementClassNames

All qualified names listed in the from or to lists of a classify statement must resolve to classes.

[5] classifyStatementExpression

The expression in a classify statement must have a class as its type and multiplicity upper bound of 1.

[6] classifyStatementFromClassDerivation

The from classes of a classify statement are the class referents of the qualified names in the from list for the statement.

[7] classifyStatementToClassDerivation

The to classes of a classify statement are the class referents of the qualified names in the to list for the statement.

Helper Operations

None

14.2.8 ConcurrentClauses

A grouping of non-final conditional clauses to be tested concurrently.

Generalizations

- SyntaxElement

Synthesized Properties

- clause : NonFinalClause [1..*]

  The conditional clauses in the group.

Derived Properties

- assignmentBefore : AssignedSource [*]

  The assigned sources for local names available lexically before this group of conditional clauses.

Constraints

[1] concurrentClausesAssignmentsBefore

The assignments before the condition of each of the clauses in a set of concurrent clauses are the same as the assignments before the concurrent clauses.

[2] concurrentClausesConditionAssignments

The same name may not be assigned in more than one conditional expression within the same concurrent set of clauses.

Helper Operations

None

14.2.9 DoStatement

A looping statement for which the continuation condition is first tested after the first iteration.

Generalizations

- Statement

Synthesized Properties

- body : Block

  The sequence of statements to be iteratively executed.

- condition : Expression

  The expression to be evaluated to determine whether to continue looping.

Derived Properties

None

Constraints

[1] doStatementAssignmentsAfter

If the assigned source for a name after the condition expression is different than before the do statement, then the assigned source of the name after the do statement is the do statement. Otherwise it is the same as before the do statement. The assignments after the do statement are adjusted for known null and non-null names and type classifications due to the condition expression being false.

[2] doStatementAssignmentsBefore

The assignments before the block of a do statement are the same as the assignments before the do statement, except that any local names with a multiplicity lower bound of 0 after the condition expression are adjusted to also have a multiplicity lower bound of 0 before the block.. The assignments before the condition expression of a do statement are the same assignments after the block.

[3] doStatementCondition

The condition expression of a do statement must have a type that conforms to type Boolean and multiplicity [1..1].

[4] doStatementEnclosedStatements

The enclosing statement for all statements in the body of a do statement are the do statement.

Helper Operations

None

14.2.10 EmptyStatement

A statement that has no affect when executed.

Generalizations

- Statement

Synthesized Properties

None

Derived Properties

None

Constraints

[1] emptyStatementAssignmentsAfter

The assignments after and empty statement are the same as the assignments before the statement.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

An empty statement may not have any annotations.

14.2.11 ExpressionStatement

A statement that evaluates an expression when executed.

Generalizations

- Statement

Synthesized Properties

- expression : Expression

  The expression to be evaluated.

Derived Properties

None

Constraints

[1] expressionStatementAssignmentsAfter

The assignments after an expression statement are the same as the assignments after its expression.

[2] expressionStatementAssignmentsBefore

The assignments before the expression of an expression statement are the same as the assignments before the statement.

Helper Operations

None

14.2.12 ForStatement

A looping statement that gives successive values to one or more loop variables on each iteration.

Generalizations

- Statement

Synthesized Properties

- body : Block

  The sequence of statements to be iteratively executed.

- variableDefinition : LoopVariableDefinition [1..*]

  A list of definitions of one or more loop variables.

Derived Properties

- isParallel : Boolean

  Whether the for statement is parallel.

Constraints

[1] forStatementAssignmentsAfter

The loop variables are unassigned after a for statement. Other than the loop variables, if the assigned source for a name after the body of a for statement is the same as after the for variable definitions, then the assigned source for the name after the for statement is the same as after the for variable definitions. If name has a different assigned source after the body of the for statement than after the for variable definitions, then the assigned source after the for statement is the for statement itself.

[2] forStatementAssignmentsBefore

The assignments before a loop variable definition in a for statement are the same as before the for statement. The assignments before the body of the statement include all the assignments before the statement plus any new assignments from the loop variable definitions, except that, if the statement is parallel, the assigned sources of any names given in @parallel annotations are changed to be the for statement itself.

[3] forStatementEnclosedStatements

The enclosing statement for all statements in the body of a for statement are the for statement.

[4] forStatementIsParallelDerivation

A for statement is parallel if it has a @parallel annotation.

[5] forStatementLoopVariables

The assigned sources for loop variables after the body of a for statement must be the for statement (the same as before the body).

[6] forStatementParallelAnnotationNames

A @parallel annotation of a for statement may include a list of names. Each such name must be already assigned after the loop variable definitions of the for statement, with a multiplicity upper bound other than 1. These names may only be used within the body of the for statement as the first argument for the

CollectionFunctions::add behavior.

[7] forStatementParallelAssignmentsAfter

If, after the loop variable definitions of a parallel for statement, a name has an assigned source, then it must have the same assigned source after the block of the for statement. Other than for names defined in the @parallel annotation of the for statement, the assigned source for such names is the same after the for statement as before it. Any names defined in the @parallel annotation have the for statement itself as their assigned source after the for statement. Other than names given in the @parallel annotation, if a name is unassigned after the loop variable definitions, then it is considered unassigned after the for statement, even if it is assigned in the block of the for statement.

[8] forStatementVariableDefinitions

The isFirst attribute of the first loop variable definition for a for statement is true while the isFirst attribute if false for any other definitions.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

In addition to an @isolated annotation, a for statement may have a @parallel annotation.

14.2.13 IfStatement

A conditional statement that executes (at most) one of a set of clauses based on boolean conditions.

Generalizations

- Statement

Synthesized Properties

- finalClause : Block [0..1]

  A sequence of statements to be executed if no other clause has a successful condition.

- nonFinalClauses : ConcurrentClauses [1..*]

  A list of groupings of concurrent clauses that are to be checked sequentially for a successful condition.

Derived Properties

- isAssured : Boolean

  Whether at least one condition in the if statement is assured to evaluate to true.

- isDeterminate : Boolean

  Whether at most one condition in the if statement will ever to evaluate to true.

Constraints

[1] ifStatementAssignmentsAfter

Any name that is unassigned before an if statement and is assigned in one or more clauses of the if statement, has, after the if statement, a type that is is the effective common ancestor of the types of the name in each clause in which it is defined. For a name that has an assigned source after any clause of an if statement that is different than before that clause, then the assigned source after the if statement is the

if statement, with a multiplicity lower bound that is the minimum of the lower bound for the name in each clause and a multiplicity upper bound that is the maximum for the name in each clause (where the name is considered to have multiplicity [0..0] for clauses in which it is not defined and unchanged multiplicity for an implicit final clause, unless the if statement is assured). Otherwise, the assigned source of a name after the if statement is the same as before the if statement.

[2] ifStatementAssignmentsBefore

The assignments before each non-final clause of an if statement are the same as the assignments before the if statement, adjusted for known nulls and non-nulls and type classifications due to the failure of the conditions in all previous sets of concurrent clauses. If the statement has a final clause, then the assignments before that clause are also the same as the assignments before the if statement, adjusted for the failure of the conditions of all previous clauses.

[3] ifStatementEnclosedStatements

The enclosing statement of all the statements in the bodies of all non-final clauses and in the final clause (if any) of an if statement is the if statement.

[4] ifStatementIsAssuredDerivation

An if statement is assured if it has an @assured annotation.

[5] ifStatementIsDeterminateDerivation

An if statement is determinate if it has an @determinate annotation.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

In addition to an @isolated annotation, an if statement may have @assured and @determinate annotations. They may not have arguments.

[2] hasReturnValue( ) : Boolean

An if statement has a return value if the bodies of all its clauses have return values, and it either has a final clause or is assured.

14.2.14 InLineStatement

A statement that executes code in a language other than Alf.

Generalizations

- Statement

Synthesized Properties

- code : String

  The in-line code to be executed.

- language : String

  The name of the language in which the code is written.

Derived Properties
None

Constraints
[1] inLineStatementAssignmentsAfter

The assignments after an in-line statement are the same as the assignments before the statement.

Helper Operations
None

14.2.15 LocalNameDeclarationStatement
A statement that declares the type of a local name and assigns it an initial value.

Generalizations
- Statement

Synthesized Properties
- expression : Expression
  The expression to be evaluated to provide the initial value to be assigned to the local name.

- hasMultiplicity : Boolean = false
  Whether the local name is to have a multiplicity upper bound of * rather than 1.

- name : String
  The local name being declared.

- typeName : QualifiedName [0..1]
  The declared type of the local name.

Derived Properties
- type : ElementReference [0..1]
  The type declared for the given local name.

Constraints
[1] localNameDeclarationStatementAssignmentsAfter

The assignments after a local name declaration statement are the assignments after the expression of the statement plus a new assignment for the local name defined by the statement. The assigned source for the local name is the local name declaration statement. The local name has the type denoted by the type name if this is not empty and is untyped otherwise. The multiplicity lower bound of the local name is 0 if the expression has a lower bound of 0, otherwise it is 1. If the statement has multiplicity, then the multiplicity upper bound of the local name is *, otherwise it is 1.

[2] localNameDeclarationStatementAssignmentsBefore

The assignments before the expression of a local name declaration statement are the same as the assignments before the statement.

[3] localNameDeclarationStatementExpressionMultiplicity

If a local name declaration statement does not have multiplicity, then the multiplicity of upper bound of the assigned expression must not be greater than 1.

[4] localNameDeclarationStatementExpressionType

If the expression of a local name declaration statement is an instance creation expression with no constructor, and the type of the statement is a class or (structured) data type, then the referent of the expression is the type of the statement. If the expression of a local name declaration statement is a sequence construction expression with no type name, but with non-empty elements, then the type of the expression is the type of the statement and the expression has multiplicity if and only if the statement does.

[5] localNameDeclarationStatementLocalName

The local name in a local name declaration statement must be unassigned before the statement and before the expression in the statement. It must remain unassigned after the expression.

[6] localNameDeclarationStatementType

If the type name in a local name declaration statement is not empty, then it must resolve to a non-template classifier and the expression must be assignable to that classifier.

[7] localNameDeclarationStatementTypeDerivation

The type of a local name declaration statement with a type name is the single classifier referent of the type name. Otherwise the type is empty.

Helper Operations

None

14.2.16 LoopVariableDefinition

The definition of a loop variable in a for statement.

Generalizations

- SyntaxElement

Synthesized Properties

- expression1 : Expression

  If there is only one expression, then this expression is evaluated to produce a sequence of values to be assigned to the loop variable on successive iterations. Otherwise it is evaluated to provide the first value of a range of values to be assigned to the loop variable.

- expression2 : Expression [0..1]

  The expression to be evaluated to give the second value in a range of values to be assigned to the loop variable.

- typeIsInferred : Boolean = true

  Whether the type of the variable is inferred or declared explicitly.

NOTE: This flag is necessary to because a variable that is explicitly declared to have type "any" will have an empty typeName, just like a variable whose type is to be inferred, but, in the former case, the type is actually intended to be empty, not inferred.

- typeName : QualifiedName [0..1]
  The declared type of the loop variable.

- variable : String
  The name of the loop variable.

Derived Properties

- assignmentAfter : AssignedSource [*]
  The assigned sources for local names available lexically after this loop variable definition. This includes not only any assignments made within the statement, but also any assignments that are unchanged from before the statement.
- assignmentBefore : AssignedSource [*]
  The assigned sources for local names available lexically before this loop variable definition.

- isCollectionConversion : Boolean
  Whether collection conversion is required.

- isFirst : Boolean
  Whether this definition is the first in the list of definitions in the enclosing for statement.

- type : ElementReference [0..1]
  The declared or inferred type of the loop variable.

Constraints

[1] loopVariableDefinitionAssignmentAfterDerivation

The assignments after a loop variable definition include the assignments after the expression (or expressions) of the definition plus a new assigned source for the loop variable itself. The assigned source for the loop variable is the loop variable definition. The multiplicity upper bound for the variable is 1. The multiplicity lower bound is 1 if the loop variable definition is the first in a for statement and 0 otherwise. If collection conversion is not required, then the variable has the inferred or declared type from the definition. If collection conversion is required, then the variable has the sequence type of the collection class.

[2] loopVariableDefinitionAssignmentsBefore

The assignments before the expressions of a loop variable definition are the assignments before the loop variable definition.

[3] loopVariableDefinitionDeclaredType

If the type of a loop variable definition is not inferred, then the first expression of the definition must have a type that conforms to the declared type.

[4] loopVariableDefinitionIsCollectionConversionDerivation

Collection conversion is required for a loop variable definition if the type for the definition is a collection class and the multiplicity upper bound of the first expression is no greater than 1.

[5] loopVariableDefinitionRangeExpressions

If a loop variable definition has two expressions, then both expressions must have type Integer and a multiplicity upper bound of 1, and no name may be newly assigned or reassigned in more than one of the expressions.

[6] loopVariableDefinitionTypeDerivation

If the type of a loop variable is not inferred, then the variable has the type denoted by the type name, if it is not empty, and is untyped otherwise. If the type is inferred, them the variable has the same as the type of the expression in its definition.

[7] loopVariableDefinitionTypeName

If a loop variable definition has a type name, then this name must resolve to a non-template classifier.

[8] loopVariableDefinitionVariable

The variable name given in a loop variable definition must be unassigned after the expression or expressions in the definition.

Helper Operations

None

14.2.17 NonFinalClause

A clause of an if statement with a conditional expression and a sequence of statements that may be executed if the condition is true.

Generalizations

- SyntaxElement

Synthesized Properties

- body : Block

  The sequence of statements that may be executed if the condition evaluates to true.

- condition : Expression

  The expression that is evaluated to determine whether the clause body may be executed.

Derived Properties

None

Constraints

[1] nonFinalClauseAssignmentsBeforeBody

The assignments before the body of a non-final clause are the assignments after the condition, adjusted for known null and non-null names and type classifications due to the condition being true.

[2] nonFinalClauseConditionLocalNames

If a name is unassigned before the condition expression of a non-final clause, then it must be unassigned after that expression (i.e., new local names may not be defined in the condition).

[3] nonFinalClauseConditionType

The condition of a non-final clause must have a type that conforms to type Boolean and multiplicity [1..1].

Helper Operations

[1] assignmentsAfter ( ) : AssignedSource [*]

The assignments after a non-final clause are the assignments after the block of the clause.

[2] assignmentsBefore ( ) : AssignedSource [*]

The assignments before a non-final clause are the assignments before the condition of the clause.

14.2.18 QualifiedNameList

A group of qualified names.

Generalizations

- SyntaxElement

Synthesized Properties

- name : QualifiedName [1..*]

  The names in the group.

Derived Properties

None

Constraints

None

Helper Operations

None

14.2.19 ReturnStatement

A statement that provides a value for the return parameter of an activity.

Generalizations

- Statement

Synthesized Properties

- expression : Expression [0..1]

  The expression to be evaluated to provide the returned value.

Derived Properties

- behavior : ElementReference

  A reference to the enclosing behavior for this return statement.

Constraints

[1] returnStatementAssignmentsAfter

The assignments after a return statement are the same as the assignments after the expression of the return statement.

[2] returnStatementAssignmentsBefore

The assignments before the expression of a return statement are the same as the assignments before the statement.

[3] returnStatementContext

If the behavior containing the return statement has a return parameter, then the return statement must have an expression, and the expression must be assignable to that return parameter.

Helper Operations

[1] hasReturnValue( ) : Boolean

A return statement is considered to have a return value.

14.2.20 Statement

A model of an Alf statement.

Generalizations

- DocumentedElement

Synthesized Properties

- annotation : Annotation [*]

  The annotations applied to this statement.

Derived Properties

- assignmentAfter : AssignedSource [*]

  The assigned sources for local names available lexically after this statement. This includes not only any assignments made within the statement, but also any assignments that are unchanged from before the statement.

- assignmentBefore : AssignedSource [*]

  The assigned sources for local names available lexically before this statement.

- enclosingStatement : Statement [0..1]

  The statement immediately enclosing this statement, if any.

- isIndexFrom0 : Boolean

  Whether indexing should be from 0 within this statement.

- isIsolated : Boolean
  Whether this statement should be executed in isolation.

Constraints

[1] statementAnnotationsAllowed

All the annotations of a statement must be allowed, as given by the annotationAllowed operation for the statement.

[2] statementIsIndexFrom0Derivation

A statement has indexing from 0 if it has an @indexFrom0 annotation, or it is contained in a statement with indexing from 0 and it does not have an @indexFrom1 annotation applied.

[3] statementIsIsolatedDerivation

A statement is isolated if it has an @isolated annotation.

[4] statementUniqueAssignments

No name may be assigned more than once before or after a statement.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

Returns true if the given annotation is allowed for this kind of statement. By default, only @isolated, @indexFrom0 and @indexFrom1 annotations are allowed, with no arguments. This operation is redefined only in subclasses of Statement for kinds of statements that allow different annotations than this default.

[2] hasReturnValue( ) : Boolean

Returns true if this statement is assured to generate a return value. By default, a statement does not have a return value.

14.2.21 SwitchClause

A clause in a switch statement with a set of cases and a sequence of statements that may be executed if one of the cases matches the switch value.

Generalizations

- SyntaxElement

Synthesized Properties

- block : Block
  The sequence of statements that may be executed if one of the cases matches the switch value.

- case : Expression [1..*]
  The expressions to be evaluated to provide values to test against the switch value.

Derived Properties

None

Constraints

[1] switchClauseAssignmentsBefore

The assignments before the case expressions of any case expression of a switch clause are the same as the assignments before the clause. The assignments before the block of a switch clause are the assignments after all case expressions.

[2] switchClauseCaseLocalNames

If a name is unassigned before a switch clause, then it must be unassigned after all case expressions of the clause (i.e., new local names may not be defined in case expressions).

[3] switchClauseCases

All the case expressions of a switch clause must have a multiplicity no greater than 1.

Helper Operations

[1] assignmentsAfter ( ) : AssignedSource [*]

The assignments after a switch clause are the assignments after the block of the switch clause.

[2] assignmentsBefore ( ) : AssignedSource [*]

The assignments before a switch clause are the assignments before any case expression of the clause.

14.2.22 SwitchStatement

A statement that executes (at most) one of a set of statement sequences based on matching a switch value to a set of test cases.

Generalizations

- Statement

Synthesized Properties

- defaultClause : Block [0..1]

  A sequence of statements to be executed if no switch clause case matches the switch value.

- expression : Expression

  The expression to be evaluated to provide the switch value.

- nonDefaultClause : SwitchClause [*]

  The set of switch clauses whose cases are to be tested against the switch value.

Derived Properties

- isAssured : Boolean

  Whether at least one case in the switch statement is assured to match.

- isDeterminate : Boolean

  Whether at most one case in the if statement will ever to match.

Constraints

[1] switchStatementAssignments

Any name that is unassigned before a switch statement and is assigned in one or more clauses of the switch statement, has, after the switch statement, a type that is is the effective common ancestor of the types of the name in each clause in which it is defined.

[2] switchStatementAssignmentsAfter

If a name has an assigned source after any clause of a switch statement that is different than before that clause (including newly defined names), the assigned source after the switch statement is the switch statement, with a multiplicity lower bound that is the minimum of the lower bound for the name in each clause and a multiplicity upper bound that is the maximum for the name in each clause (where the name is considered to have multiplicity [0..0] for clauses in which it is not defined and unchanged multiplicity for an implicit default clause, unless the switch statement is assured). Otherwise, the assigned source of a name after the switch statement is the same as before the switch statement.

[3] switchStatementAssignmentsBefore

The assignments before all clauses of a switch statement are the same as the assignments after the expression of the switch statement.

[4] switchStatementCaseAssignments

The same local name may not be assigned in more than one case expression in a switch statement.

[5] switchStatementEnclosedStatements

A switch statement is the enclosing statement for the statements in all of its switch clauses.

[6] switchStatementExpression

A switch statement expression must have a multiplicity no greater than 1.

[7] switchStatementIsAssuredDerivation

A switch statement is assured if it has an @assured annotation.

[8] switchStatementIsDeterminateDerivation

A switch statement is determinate if it has a @determinate annotation.

Helper Operations

[1] annotationAllowed (in annotation : Annotation) : Boolean

In addition to an @isolated annotation, a switch statement may have @assured and @determinate annotations. They may not have arguments.

[2] hasReturnValue( ) : Boolean

A switch statement has a return value if the blocks of all its clauses have return values, and it either has a default clause or is assured.

14.2.23 WhileStatement

A looping statement for which the continuation condition is first tested before the first iteration.

Generalizations

- Statement

Synthesized Properties

- body : Block

  The sequence of statements to be iteratively executed.

- condition : Expression

  The expression to be evaluated to determine whether to continue looping.

Derived Properties

None

Constraints

[1] whileStatementAssignmentsAfter

If the assigned source for a name after the block of a while statement is different than before the while statement, then the assigned source of the name after the while statement is the while statement. Otherwise it is the same as before the while statement. If a name is unassigned before the block of a while statement and assigned after the block, then it has multiplicity lower bound of 0 after the while statement. Otherwise, the assignments after the while statement are adjusted for known null and non-null names and type classifications due to the condition expression being false.

[2] whileStatementAssignmentsBefore

The assignments before the condition expression of a while statement are the same as the assignments before the while statement, except that any local names with a multiplicity lower bound of 0 after the block are adjusted to also have a multiplicity lower bound of 0 before the condition expression. The assignments before the block of the while statement are the same as the assignments after the condition expression, adjusted for known null and non-null names and type classifications due to the condition expression being true.

[3] whileStatementCondition

The condition expression of a while statement must have a type that conforms to type Boolean and multiplicity [1..1].

[4] whileStatementEnclosedStatements

The enclosing statement for all statements in the body of a while statement are the while statement.

Helper Operations

None

15 Units Abstract Syntax

15.1 Overview

The `Alf::Syntax::Units` package contains the abstract syntax model for units. The syntax and semantics of statements are discussed in Clause 10. Their mapping to UML is given in Clause 19.

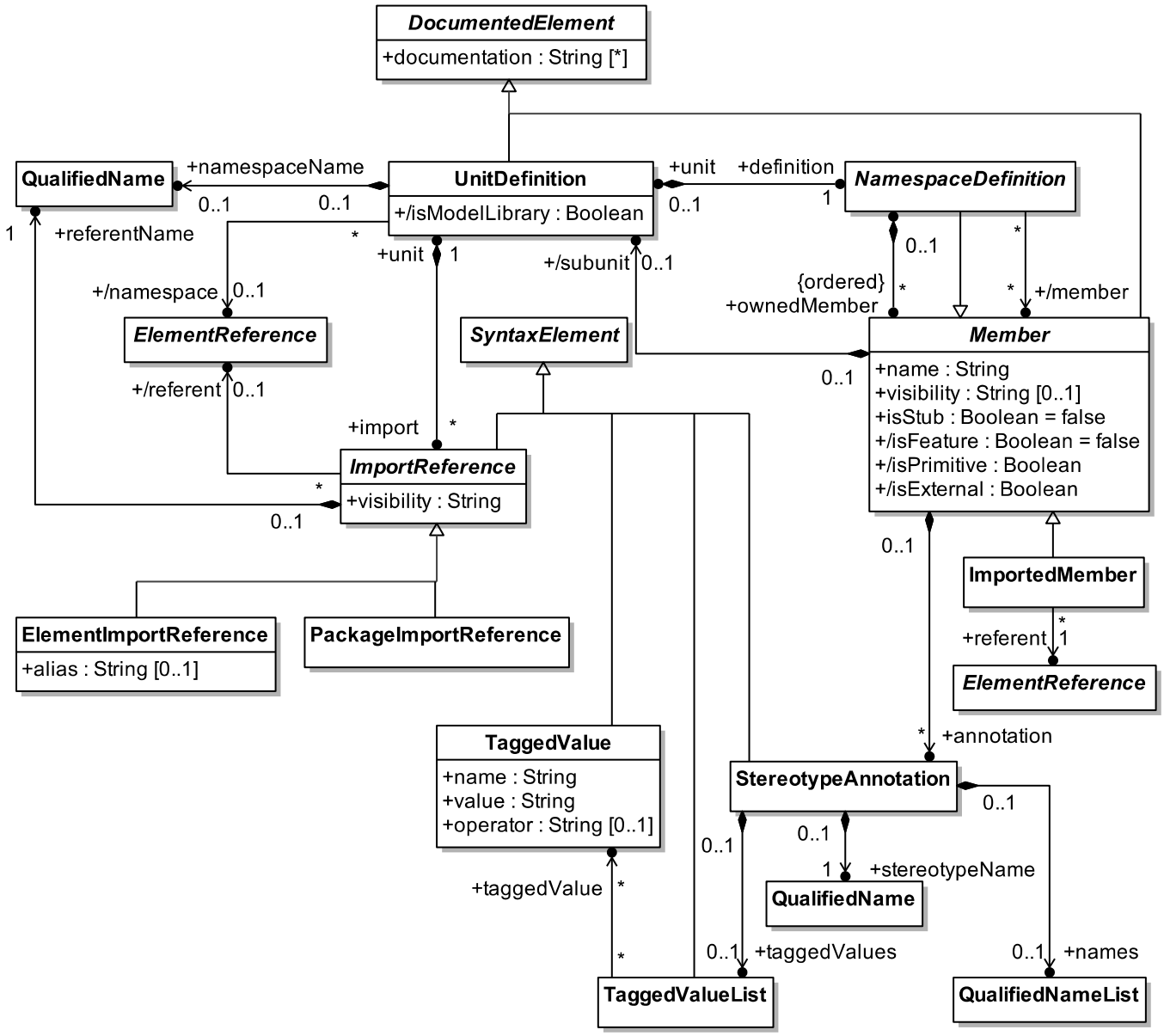

Figure 15.1 Unit and Namespace Definitions

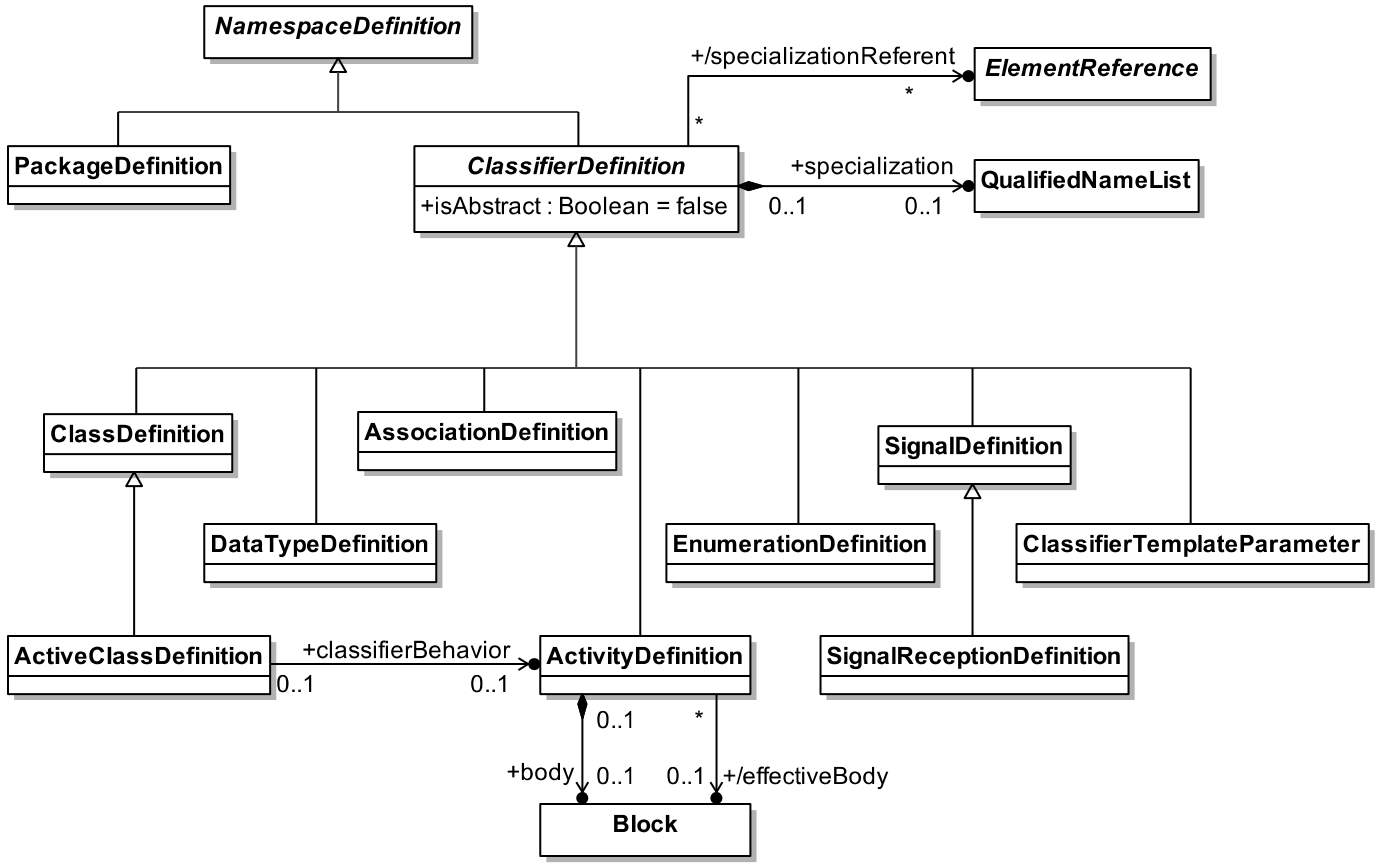

Figure 15.2 Package and Classifier Definitions

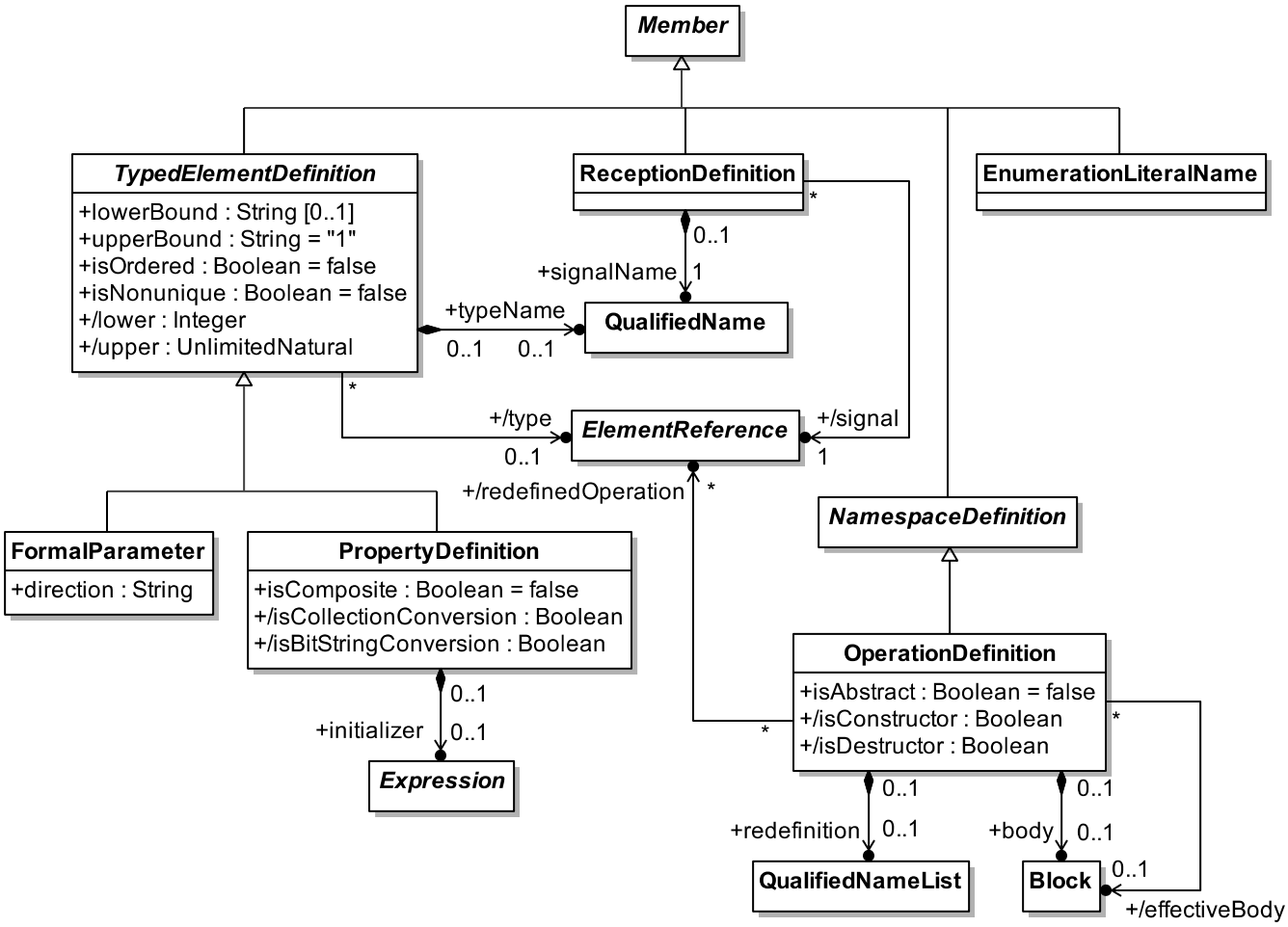

Figure 15.3 Parameter, Feature and Enumeration Literal Definitions

15.2 Class Descriptions

15.2.1 ActiveClassDefinition

The definition of an active class.

Generalizations

- ClassDefinition

Synthesized Properties

- classifierBehavior : ActivityDefinition [0..1]

  The definition of an activity (which may be a stub) to act as the classifier behavior of the active class.

Derived Properties

None

Constraints

[1] activeClassDefinitionClassifierBehavior

If an active class definition is not abstract, then it must have a classifier behavior.

Helper Operations

[1] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this active class definition considered as a class definition and the subunit is for an active class definition.

15.2.2 ActivityDefinition

The definition of an activity, with any formal parameters defined as owned members.

Generalizations

- ClassifierDefinition

Synthesized Properties

- body : Block [0..1]

  The sequence of statements that defines the behavior of the activity (empty for a stub).

Derived Properties

- effectiveBody : Block [0..1]
  If this activity definition is a stub, then the body of the corresponding subunit.

Constraints

[1] activityDefinitionEffectiveBodyAssignmentsBefore

The assignments before the effective body of an activity definition include an assignment for each "in" or "inout" formal parameter of the activity definition, with the formal parameter as the assigned source.

[2] activityDefinitionEffectiveBodyDerivation

If an activity definition is a stub, then its effective body is the body of the corresponding subunit. Otherwise, the effective body is the same as the body of the activity definition.

[3] activityDefinitionPrimitive

If an activity definition is primitive, then it must have a body that is empty.

[4] activityDefinitionReturn

If an activity definition has a return parameter with a multiplicity lower bound greater than 0, then the effective body of the activity definition must have a return value.

[5] activityDefinitionSpecialization

An activity definition may not have a specialization list.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, an activity definition allows @primitive annotations and any stereotype whose metaclass is consistent with Activity.

[2] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this activity definition considered as a classifier definition and the subunit is for an activity definition. In addition, the subunit definition must have formal parameters that match each of the formal parameters of the stub definition, in order. Two formal parameters match if they have the same direction, name, multiplicity bounds, ordering, uniqueness and type reference.

15.2.3 AssociationDefinition

The definition of an association, whose members must all be properties.

Generalizations

- ClassifierDefinition

Synthesized Properties

None

Derived Properties

None

Constraints

[1] associationDefinitionSpecializationReferent

The specialization referents of an association definition must all be associations.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, an association definition allows an annotation for any stereotype whose metaclass is consistent with Association.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either an AssociationDefinition or an imported member whose referent is an AssociationDefinition or an Association.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this association definition considered as a classifier definition and the subunit is for an association definition.

15.2.4 ClassDefinition

The definition of a class, whose members may be properties, operations, signals or signal receptions.

Generalizations

- ClassifierDefinition

Synthesized Properties
None

Derived Properties
None

Constraints
[1] classDefinitionAbstractMember

If a class definition is not abstract, then no member operations (owned or inherited) of the class definition may be abstract.

[2] classDefinitionSpecializationReferent

The specialization referents of a class definition must all be classes. A class definition may not have any referents that are active classes unless this is an active class definition.

Helper Operations
[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, a class definition allows an annotation for any stereotype whose metaclass is consistent with Class.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a ClassDefinition or an imported member whose referent is a ClassDefinition or a Class.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this class definition considered as a classifier definition and the subunit is for a class definition.

15.2.5 ClassifierDefinition
The definition of a classifier.

Generalizations
- NamespaceDefinition

Synthesized Properties
- isAbstract : Boolean = false

  Whether the classifier is abstract or not.

- specialization : QualifiedNameList [0..1]

  The names of classifiers specialized by the classifier being defined.

Derived Properties
- specializationReferent : ElementReference [*]

  References to the classifiers to which the names in the specialization list resolve.

Constraints

[1] classifierDefinitionInheritedMembers

The members of a classifier definition include non-private members inherited from the classifiers it specializes. The visibility of inherited members is as specified in the UML Superstructure, 7.3.8. Elements inherited from external classifiers are treated as imported members.

[2] classifierDefinitionSpecialization

Each name listed in the specialization list for a classifier definition must have a single classifier referent. None of these referents may be templates.

[3] classifierDefinitionSpecializationReferentDerivation

The specialization referents of a classifier definition are the classifiers denoted by the names in the specialization list for the classifier definition.

Helper Operations

[1] matchForStub (in unit : UnitDefinition) : Boolean

The namespace definition associated with the given unit definition must be a classifier definition. The subunit classifier definition may be abstract if and only if the subunit classifier definition is abstract. The subunit classifier definition must have the same specialization referents as the stub classifier definition. (Note that it is the referents that must match, not the exact names or the ordering of those names in the specialization list.) The subunit classifier definition must also have a matching classifier template parameter for each classifier template parameter of the stub classifier definition. Two template parameters match if they have same names and the same specialization referents.

15.2.6 ClassifierTemplateParameter

The definition of a classifier template parameter, which acts as a classifier within the definition of the template.

Generalizations

- ClassifierDefinition

Synthesized Properties

None

Derived Properties

None

Constraints

None

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Annotations are not allowed on classifier template parameters.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is a classifier template parameter.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns false. (Classifier template parameters cannot be stubs.)

15.2.7 DataTypeDefinition

The definition of a data type, whose members must all be properties.

Generalizations

- ClassifierDefinition

Synthesized Properties

None

Derived Properties

None

Constraints

[1] dataTypeDefinitionPrimitive

If a data type is primitive, then it may not have any owned members.

[2] dataTypeDefinitionSpecializationReferent

The specialization referents of a data type definition must all be data types.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, a data type definition allows @primitive annotations plus any stereotype whose metaclass is consistent with DataType.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a DataTypeDefinition or an imported member whose referent is a DataTypeDefinition or a DataType.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this data type definition considered as a classifier definition and the subunit is for a data type definition.

15.2.8 ElementImportReference

An import reference to a single element to be imported into a unit.

Generalizations

- ImportReference

Synthesized Properties

- alias : String [0..1]

  The alias to be used as the name for the imported element in the importing unit's namespace.

Derived Properties

None

Constraints

[1] elementImportReferenceReferent

The referent of an element import reference must be a packageable element.

Helper Operations

None

15.2.9 EnumerationDefinition

The definition of an enumeration, whose members must all be enumeration literal names.

Generalizations

- ClassifierDefinition

Synthesized Properties

None

Derived Properties

None

Constraints

[1] enumerationDefinitionSpecializationReferent

The specialization referents of a class definition must all be classes. A class definition may not have any referents that are active classes unless this is an active class definition.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, an enumeration definition allows an annotation for any stereotype whose metaclass is consistent with Enumeration.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either an EnumerationDefinition or an imported member whose referent is an EnumerationDefinition or an Enumeration.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this enumeration definition considered as a classifier definition and the subunit is for an enumeration definition.

15.2.10 EnumerationLiteralName

The definition of an enumeration literal, as a member of an enumeration definition.

Generalizations

- Member

Synthesized Properties

None

Derived Properties
None

Constraints
None

Helper Operations
[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns false. (Enumeration literal name cannot have annotations.)

15.2.11 FormalParameter

A typed element definition for the formal parameter of an activity or operation.

Generalizations
- TypedElementDefinition

Synthesized Properties
- direction : String

  An indication of the direction of the parameter being defined.

Derived Properties
None

Constraints
[1] formalParameterAssignmentAfterBody
If a formal parameter has direction "out" and a multiplicity lower bound greater than 0, and its owning activity or operation definition has an effective body, then there must be an assignment for the formal parameter after the effective body that has a multiplicity greater than 0.

Helper Operations
[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true if the annotation is for a stereotype that has a metaclass consistent with Parameter.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is a FormalParameter.

15.2.12 ImportedMember

Generalizations
- Member

Synthesized Properties
- referent : ElementReference

Derived Properties
None

Constraints

[1] importedMemberIsFeatureDerivation

An imported element is a feature if its referent is a feature.

[2] importedMemberNotStub

An imported element is not a stub.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns false. (Imported members do not have annotations.)

[2] isSameKindAs (in member : Member) : Boolean

If the given member is not an imported member, then return the result of checking whether the given member is the same kind as this member. Else, if the element of the referent for this member is an Alf member, then return the result of checking whether that element is the same kind as the given member. Else, if the element of the referent for the given member is an Alf member, then return the result of checking whether that element is the same kind as this member. Else, the referents for both this and the given member are UML elements, so return the result of checking their distinguishability according to the rules of the UML superstructure.

15.2.13 ImportReference

A reference to an element or package to be imported into a unit.

Generalizations

- SyntaxElement

Synthesized Properties

- referentName : QualifiedName

  The name of the element or package to be imported.

- unit : UnitDefinition

  The unit that is making this import reference.

- visibility : String

  An indication of the visibility of the import.

Derived Properties

- referent : ElementReference [0..1]

  A reference to the imported element denoted by the given qualified name.

Constraints

[1] importReferenceReferent

The referent name of an import reference must resolve to a single element with public or empty visibility.

[2] importReferenceReferentDerivation

The referent of an import reference is the element denoted by the referent name.

Helper Operations

None

15.2.14 Member

A model of the common properties of the definition of a member of a namespace in Alf.

Generalizations

- DocumentedElement

Synthesized Properties

- annotation : StereotypeAnnotation [*]

  The stereotype annotations on this member definition.

- isStub : Boolean = false

  Whether this member definition is a stub for a subunit.

- name : String

  The name of the member.

- namespace : NamespaceDefinition [0..1]

  The namespace definition within which this member definition is nested, if any. (The namespace definitions for units are not physically nested within another Alf namespace definition.)

- visibility : String [0..1]

  An indication of the visibility of the member outside of its namespace.

Derived Properties

- isExternal : Boolean

  Whether this member is external or not.

- isFeature : Boolean = false

  Whether this member is a feature of a classifier.

- isPrimitive : Boolean

  Whether this member is a primitive or not.

- subunit : UnitDefinition [0..1]

  The subunit corresponding to the member, if the member is a stub.

Constraints

[1] memberAnnotations

All stereotype annotations for a member must be allowed, as determined using the stereotypeAllowed operation.

[2] memberExternal

If a member is external then it must be a stub.

[3] memberIsExternalDerivation

A member is external if it has an @external derivation.

[4] memberIsPrimitiveDerivation

A member is primitive if it has a @primitive annotation.

[5] memberPrimitive

If a member is primitive, then it may not be a stub and it may not have any owned members that are template parameters.

[6] memberStub

If a member is a stub and is not external, then there must be a single subunit with the same qualified name as the stub that matches the stub, as determined by the matchForStub operation.

[7] memberStubStereotypes

If a member is a stub, then the it must not have any stereotype annotations that are the same as its subunit. Two stereotype annotations are the same if they are for the same stereotype.

[8] memberSubunitDerivation

If the member is a stub and is not external, then its corresponding subunit is a unit definition with the same fully qualified name as the stub.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true of the given stereotype annotation is allowed for this kind of element.

[2] isDistinguishableFrom (in member : Member) : Boolean

Returns true if this member is distinguishable from the given member. Two members are distinguishable if their names are different or the they are of different kinds (as determined by the isSameKindAs operation). However, in any case that the UML Superstructure considers two names to be distinguishable if they are different, an Alf implementation may instead impose the stronger requirement that the names not be conflicting.

[3] isSameKindAs (in member : Member) : Boolean

Returns true if this member is of the same kind as the given member.

[4] matchForStub (in unit : UnitDefinition) : Boolean

Returns true of the given unit definition is a legal match for this member as a stub. By default, always returns false.

15.2.15 NamespaceDefinition

A model of the common properties of the definition of a namespace in Alf.

Generalizations

- Member

Synthesized Properties

- ownedMember : Member [*] {ordered}

  The definitions of owned members of the namespace.

- unit : UnitDefinition [0..1]

  The unit for which this namespace is a definition, if any.

Derived Properties

- member : Member [*]

  The owned and imported members of a namespace definition.

Constraints

[1] namespaceDefinitionMemberDerivation

The members of a namespace definition include references to all owned members. Also, if the namespace definition has a unit with imports, then the members include imported members with referents to all imported elements. The imported elements and their visibility are determined as given in the UML Superstructure. The name of an imported member is the name of the imported element or its alias, if one has been given for it. Elements that would be indistinguishable from each other or from an owned member (as determined by the Member::isDistinguishableFrom operation) are not imported.

[2] namespaceDefinitionMemberDistinguishability

The members of a namespace must be distinguishable as determined by the Member::isDistinguishableFrom operation.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true if the annotation is @external.

15.2.16 OperationDefinition

The definition of an operation, with any formal parameters defined as owned members.

Generalizations

- NamespaceDefinition

Synthesized Properties

- body : Block [0..1]

The sequence of statements that defines the behavior of the operation (empty for a stub or abstract operation).

- isAbstract : Boolean = false
  Whether the operation being defined is abstract.

- redefinition : QualifiedNameList [0..1]
  The names of other operations that are redefined by the operation being defined.

Derived Properties

- effectiveBody : Block [0..1]
  If this operation definition is a stub, then the body of the corresponding subunit.

- isConstructor : Boolean
  Whether this operation definition is for a constructor.

- isDestructor : Boolean
  Whether this operation definition is for a destructor.

- redefinedOperation : ElementReference [*]

Constraints

[1] operationDefinitionAbstractOperation
If an operation definition is abstract, then its body must be empty.

[2] operationDefinitionConstructor
If an operation definition is a constructor, any redefined operation for it must also be a constructor. The body of a constructor may contain an alternative constructor invocation for another constructor in the same class or super constructor invocations for constructors in immediate superclasses.

[3] operationDefinitionConstructorDestructor
An operation definition cannot be both a constructor and a destructor.

[4] operationDefinitionDestructor
If an operation definition is a destructor, any redefined operation for it must also be a destructor.

[5] operationDefinitionEffectiveBodyAssignmentsBefore
The assignments before the effective body of an operation definition include an assignment for each "in" or "inout" formal parameter of the operation definition, with the formal parameter as the assigned source.
[6] operationDefinitionEffectiveBodyDerivation
If an operation definition is a stub, then its effective body is the body of the corresponding subunit. Otherwise, the effective body is the same as the body of the operation definition.

[7] operationDefinitionIsConstructorDerivation

An operation definition is a constructor if it has a @Create annotation.

[8] operationDefinitionIsDestructorDerivation

An operation definition is a destructor if it has a @Destroy annotation.

[9] operationDefinitionIsFeatureDerivation

An operation definition is a feature.

[10] operationDefinitionNamespace

The namespace for an operation definition must be a class definition.

[11] operationDefinitionRedefinedOperationDerivation

If an operation definition has a redefinition list, its redefined operations are the referent operations of the names in the redefinition list for the operation definition. Otherwise, the redefined operations are any operations that would otherwise be indistinguishable from the operation being defined in this operation definition.

[12] operationDefinitionRedefinedOperations

The redefined operations of an operation definition must have formal parameters that match each of the formal parameters of this operation definition, in order. Two formal parameters match if they have the same direction, name, multiplicity bounds, ordering, uniqueness and type reference.

[13] operationDefinitionRedefinition

Each name in the redefinition list of an operation definition must have a single referent that is an operation. This operation must be a non-private operation that is a member of a specialization referent of the class definition of the operation definition.

[14] operationDefinitionReturn

If an operation definition has a return parameter with a multiplicity lower bound greater than 0, then the effective body of the operation definition must have a return value.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true if the annotation is for a stereotype that has a metaclass consistent with Operation.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either an OperationDefinition or an imported member whose referent is an OperationDefinition or an Operation, and the formal parameters of this operation definition match, in order, the parameters of the other operation definition or operation. In this context, matching means the same name and type (per UML Superstructure, 7.3.5). A constructor operation without an explicit return parameter is considered to implicitly have a return parameter, following any other formal parameters, of the same type as the owner of the constructor operation.

[3] matchForStub (in unit : UnitDefinition) : Boolean

The namespace definition associated with the given unit definition must be an activity definition with no template parameters. In addition, the subunit definition must have formal parameters that match each of the formal parameters of the stub definition, in order. Two formal parameters match if they have the same direction, name, multiplicity bounds, ordering, uniqueness and type reference If this operation definition is a constructor, then it is considered to have an implicit return parameter, following any other formal parameters, with the same type as the class of the operation definition and a multiplicity of 1..1.

15.2.17 PackageDefinition

The definition of a package, all of whose members must be packageable elements.

Generalizations

- NamespaceDefinition

Synthesized Properties

None

Derived Properties

- appliedProfile : Profile [*]

  The profiles applied (directly) to this package.

Constraints

[1] packageDefinitionAppliedProfileDerivation

The applied profiles of a package definition are the profiles listed in any @apply annotations on the package.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed on any namespace definition, a package definition allows @apply annotations plus any stereotype whose metaclass is consistent with Package.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a PackageDefinition or an imported member whose referent is a PackageDefinition or a Package.

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true of the namespace definition associated with the given unit definition is a package definition.

15.2.18 PackageImportReference

An import reference to a package all of whose public members are to be imported.

Generalizations

- ImportReference

Synthesized Properties

None

Derived Properties
None

Constraints
[1] packageImportReferenceReferent

The referent of a package import must be a package.

Helper Operations
None

15.2.19 PropertyDefinition
A typed element definition for a property (attribute or association end).

Generalizations
- TypedElementDefinition

Synthesized Properties
- initializer : Expression [0..1]

  The expression to be evaluated to initialize the property.

- isComposite : Boolean = false

  Whether the property being defined has composite aggregation.

Derived Properties
- isBitStringConversion : Boolean

  Whether BitString conversion is required for the initialization of this property.

- isCollectionConversion : Boolean

  Whether collection conversion is required for the initialization of this property.

Constraints
[1] propertyDefinitionInitializer

If a property definition has an initializer, then the initializer expression must be assignable to the property definition. There are no assignments before an initializer expression.

[2] propertyDefinitionInitializerType

If the initializer of a property definition is an instance creation expression with no constructor, and the type of the property definition is a class or (structured) data type, then the referent of the expression is the type of the property definition. If the initializer of a property definition is a sequence construction expression with no type name, but with non-empty elements, then the type of the expression is the type of the property definition and the expression has multiplicity if and only if the multiplicity upper bound of the property definition is greater than 1.

[3] propertyDefinitionIsBitStringConversionDerivation

A property definition requires BitString conversion if its type is BitString and the type of its initializer is Integer or a collection class whose sequence type is Integer.

[4] propertyDefinitionIsCollectionConversionDerivation

A property definition requires collection conversion if its initializer has a collection class as its type and the property definition does not.

[5] propertyDefinitionIsFeatureDerivation

A property definition is a feature.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true if the annotation is for a stereotype that has a metaclass consistent with Property.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a PropertyDefinition or an imported member whose referent is a PropertyDefinition or a Property.

15.2.20 ReceptionDefinition

The declaration of the ability of an active class to receive a signal.

Generalizations

- Member

Synthesized Properties

- signalName : QualifiedName

  The name of the signal to be received.

Derived Properties

- signal : ElementReference

Constraints

[1] receptionDefinitionIsFeatureDerivation

A reception definition is a feature.

[2] receptionDefinitionSignalDerivation

The signal for a reception definition is the signal referent of the signal name for the reception definition.

[3] receptionDefinitionSignalName

The signal name for a reception definition must have a single referent that is a signal. This referent must not be a template.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

Returns true if the annotation is for a stereotype that has a metaclass consistent with Reception.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a ReceptionDefinition, a SignalReceptionDefinition or an imported member whose referent is a ReceptionDefinition, a SignalReceptionDefinition, or a Reception.

15.2.21 SignalDefinition

The definition of a signal, whose members must all be properties.

Generalizations

- ClassifierDefinition

Synthesized Properties

None

Derived Properties

None

Constraints

[1] signalDefinitionSpecializationReferent

The specialization referents of a signal definition must all be signals.

Helper Operations

[1] annotationAllowed (in annotation : StereotypeAnnotation) : Boolean

In addition to the annotations allowed for classifiers in general, a signal definition allows an annotation for any stereotype whose metaclass is consistent with Signal.

[2] isSameKindAs (in member : Member) : Boolean

Return true if the given member is either a SignalDefinition or an imported member whose referent is a SignalDefinition or a Signal (where signal reception definitions are considered to be kinds of signal definitions).

[3] matchForStub (in unit : UnitDefinition) : Boolean

Returns true if the given unit definition matches this signal definition considered as a classifier definition and the subunit is for a signal definition.

15.2.22 SignalReceptionDefinition

The definition of both a signal and a reception of that signal as a feature of the containing active class.

Generalizations

- SignalDefinition

Synthesized Properties

None

Derived Properties
None

Constraints
[1] signalReceptionDefinitionIsFeatureDerivation

A signal reception definition is a feature.

Helper Operations
None

15.2.23 StereotypeAnnotation
An annotation of a member definition indicating the application of a stereotype (or one of a small number of special-case annotations).

Generalizations
- SyntaxElement

Synthesized Properties
- names : QualifiedNameList [0..1]

  A set of references to model elements required for the stereotype being applied.

- stereotypeName : QualifiedName

  The name of the stereotype being applied.

- taggedValues : TaggedValueList [0..1]

  A set of tagged values for the applied stereotype.

Derived Properties
- stereotype : Stereotype [0..1]

  The stereotype denoted by the stereotype name.

Constraints
[1] stereotypeAnnotationApply

If the stereotype name of a stereotype annotation is "apply", then it must have a name list and all of the names in the list must resolve to profiles.

[2] stereotypeAnnotationExternal

If the stereotype name of a stereotype annotation is "external", then it may optionally have a single tagged value with the name "file" and no operator.

[3] stereotypeAnnotationNames

If a stereotype annotation has a stereotype and a list of names, then all the names in the list must resolve to visible model elements and the stereotype must have a single attribute with a (metaclass) type and multiplicity that are consistent with the types and number of the elements denoted by the given names.

[4] stereotypeAnnotationPrimitive

If the stereotype name of a stereotype annotation is "primitive", then it may not have tagged values or names.

[5] stereotypeAnnotationStereotypeDerivation

Unless the stereotype name is "apply", "primitive" or "external" then the stereotype for a stereotype annotation is the stereotype denoted by the stereotype name.

[6] stereotypeAnnotationStereotypeName

The stereotype name of a stereotype annotation must either be one of "apply", "primitive" or "external", or it must denote a single stereotype from a profile applied to an enclosing package. The stereotype name does not need to be qualified if there is only one applied profile with a stereotype of that name or if the there is a standard UML profile with the name.

[7] stereotypeAnnotationTaggedValues

If a stereotype annotation has a stereotype and tagged values, then the each tagged value must have the name of an attribute of the stereotype and a value that is legally interpretable for the type of that attribute.

Helper Operations

None

15.2.24 TaggedValue

An assignment of a value to an attribute of an applied stereotype.

Generalizations

- SyntaxElement

Synthesized Properties

- name : String

  The name of the stereotype attribute to be assigned a value.

- operator : String [0..1]

  For a numeric value, an optional unary plus or minus operator.

- value : String

  The string image of a literal value to be assigned to the stereotype attribute.

Derived Properties

None

Constraints

None

Helper Operations

None

15.2.25 TaggedValueList

A set of tagged values for a stereotype application.

Generalizations

- SyntaxElement

Synthesized Properties

- taggedValue : TaggedValue [*]

  The tagged values in the set.

Derived Properties

None

Constraints

None

Helper Operations

None

15.2.26 TypedElementDefinition

The common properties of the definitions of typed elements.

Generalizations

- Member

Synthesized Properties

- isNonunique : Boolean = false

  Whether the element being defined is non-unique.

- isOrdered : Boolean = false

  Whether the element being defined is ordered.

- lowerBound : String [0..1]

  The string image of the literal given to specify the lower bound of the multiplicity of the element being defined.

- typeName : QualifiedName [0..1]

  The name of the type of the element being defined.

- upperBound : String = “1”

  The string image of the literal given to specify the upper bound of the multiplicity of the element being defined.

Derived Properties

- lower : Integer

  The multiplicity lower bound of the element being defined.

- type : ElementReference [0..1]

- upper : UnlimitedNatural
  The multiplicity upper bound of the element being defined.

Constraints

[1] typedElementDefinitionLowerDerivation

If the lower bound string image of a typed element definition is not empty, then the integer lower bound is the integer value of the lower bound string. Otherwise the lower bound is equal to the upper bound, unless the upper bound is unbounded, in which case the lower bound is 0.

[2] typedElementDefinitionTypeDerivation

The type of a typed element definition is the single classifier referent of the type name.

[3] typedElementDefinitionTypeName

The type name of a typed element definition must have a single classifier referent. This referent may not be a template.

[4] typedElementDefinitionUpperDerivation

The unlimited natural upper bound value is the unlimited natural value of the uper bound string (with "*" representing the unbounded value).

Helper Operations

None

15.2.27 UnitDefinition

The definition of a namespace as an Alf unit.

Generalizations

- DocumentedElement

Synthesized Properties

- definition : NamespaceDefinition
  The definition of the unit as a namespace.

- import : ImportReference [*]
  The set of references to imported elements or packages.
- namespaceName : QualifiedName [0..1]
  A declaration of the name of the namespace that contains this unit as a subunit.

Derived Properties

- appliedProfile : Profile [*]
  The profiles applied to this unit.

- isModelLibrary : Boolean
  Whether this unit definition is for a model library or not.

- namespace : ElementReference [0..1]
  A reference to the namespace denoted by the declared namespace name for the unit, if any.

Constraints

[1] unitDefinitionAppliedProfileDerivation

The profiles applied to a unit definition include any profiles applied to the containing namespace of the unit definition. If the unit definition is for a package, then the applied profiles for the unit definition also include the applied profiles for its associated package definition.

[2] unitDefinitionImplicitImports

Unless the unit definition is a model library, it has private package import references for all the sub-packages of the Alf::Library package.

[3] unitDefinitionIsModelLibraryDerivation

A unit definition is for a model library if its associated namespace definition has a stereotype annotation for the UML standard stereotype ModelLibrary.

[4] unitDefinitionNamespace

The declared namespace name for a unit definition, if any, must resolve to a UML namespace of an Alf unit definition. If it is an Alf unit definition, then it must have a stub for this unit definition.

[5] unitDefinitionNamespaceDerivation

If a unit definition has a declared namespace name, then the containing namespace for the unit is the referent for that name.

Helper Operations

None

16 Common Mapping

16.1 General

The mapping specification for each syntactic area defines how a specific Alf abstract syntax tree substructure is mapped into a corresponding part of the fUML abstract syntax representation, in terms of the further subtrees of that structure. This may be considered as a metamodel to metamodel transformation.

1. The transformation is from the Alf abstract syntax metamodel to the fUML abstract metamodel.
2. The transformation maps the root objects from the Alf abstract syntax representation to UML elements. A well-formed Alf abstract syntax tree is always rooted in either an expression (see 12.2), a statement sequence (see 13.2) or a unit (see 14.2).

The remainder of this clause defines the mapping of the common elements contained in the `Alf::Syntax::Common` package (see Clause 12). The following three clauses give mappings for elements in the `Expressions`, `Statements` and `Units` packages.

16.2 Syntax Elements

1. The root mapping takes an Alf syntax element to a UML element. By default, this mapping is empty, but the mapping is overridden as appropriate for subclasses of SyntaxElement. In particular, such a mapping definition is provided for Expression, Block, UnitDefinition and any kind of syntax element that may be the target of an internal element reference.

16.3 Documented Elements

1. If an element includes documentation, then each documentation string maps to a comment element attached to mapping of the documented element, with the comment body given by the documentation text.

16.4 Element References

During mapping, an element reference is eventually mapped to a direct link to the referenced model element, either as directly identified by an external element reference or as the model element mapped from the syntax element identified by an internal element reference.

1. An element reference maps to a UML model element.
2. An external element reference maps to the identified model element.
3. An internal element reference maps to model element mapped from its identified syntax element.

16.5 Assigned Sources

An assigned source must ultimately map to an activity node that provides the source for an object flow used to obtain the assigned value, as determined by the mapping of the source syntax element.

1. The mapping of the local name to an activity node depends on the assigned source syntax element for that local name.

2. An assigned source object is mapped to the appropriate activity node as determined by querying the source syntax element.

17 Expressions Mapping

17.1 General

This clause defines the mapping of Alf expressions to UML. The abstract syntax for Alf expressions is described in Clause 13.

1. An Alf expression that is not contained in any other Alf text is mapped to UML as an activity with one parameter: a return parameter that gives the result of the expression.
2. Any Alf expression maps to some or all of the nodes and edges in an activity (sometimes called a *subgraph* of the activity).
3. The mapping of each kind of expression identifies the *result source* element in the mapping. This is the activity node to which an outgoing object flow may be attached in order to obtain the result of the expression. The result values of the expression correspond to the values of the sequence of object tokens produced on the flow. In some cases (such as when an expression is used in an expression statement; see 9.6) the result source element may remain unconnected. In this case the result values of the expression are lost.

17.2 Qualified Names

The formal mapping of a qualified name is given in various contexts of its use in subsequent subclauses. In general:

- When defining a named element, an unqualified name maps to the name of the named element, with the fully qualified name mapping to the qualified name of the named element.
- When referencing a named element, its (qualified) name maps to a reference to that named element.

17.3 Literal Expressions

1. A literal expression maps to a value specification action with the literal mapping to an appropriate literal primitive element. The result pin of the value specification action is the result source element for the expression.

17.4 Name Expressions

1. A name expression maps to an activity graph depending on the kind of name referenced.
2. A name expression for a local name or parameter name is mapped to an object flow. The source of the object flow is given by the assigned source for the name before the name expression. The target of the object flow is determined by the context of the use of the name expression.

   The assigned source of the name effectively also acts as the result source element for the expression. Note that, if this source is never connected (for example, if the name expression is used by itself as an expression statement), there can be no object flow and the name expression will actually not map to anything (since it will have no effect).

   If there is a structured activity node that owns (directly or indirectly) both the source and target of the object flow, then the most deeply nested such node owns the object flow. Otherwise it is owned by the enclosing activity.

3. A name expression for an enumeration literal name is mapped to a value specification action whose value is given by an instance literal specifying the given enumeration literal. The result pin of the value specification action is the result source element for the expression.
4. A name expression for a name that disambiguates to a feature reference is mapped as a property access expression consisting of that feature reference (see 8.3.6).

17.5 `this` Expressions

1. A `this` expression maps to a read self action. The result pin of the read self action is the result source element for the expression.

17.6 Property Access Expressions

NOTE. The Alf property access expression notation may be used to represent the access to a property (structural feature or opposite association end) of any kind of classifier other than a primitive type. However, the only kinds of non-primitive classifiers in the fUML subset with properties are classes, data types and signals. Therefore, a property access expression can only be mapped to fUML if the type of its collection expression is a data type, class or signal, and the semantics of the expression are formally defined only in this case.

1. A property access expression is mapped as either a single instance property access or a sequence property access.
2. A single instance property access expression for an attribute is mapped to a read structural feature action for the named structural feature. The result source element of the mapping of the target expression is connected by an object flow to the object input pin of the read structural feature action. The result pin of the action is the result source element for the property access expression.
3. A sequence property access expression is mapped as an expansion region similarly to a `collect` expression (see 17.19).

17.7 Invocation Expressions

1. An invocation expression is mapped as a behavior invocation or a feature invocation. Subclause 17.8 describes the mapping for tuples in general. Subclause 17.9 describes the mapping for behavior invocations (which also include a functional notation for reading associations). Subclause 17.10 describes the mapping for all other kinds of invocations. Note that, after static semantic analysis, a super invocation is mapped as a behavior invocation (see 17.11).
2. If the invocation expression is the assigned source for a local name, then it must map to a call action with result output pins. The actual source for the value of the local name is the fork node connected to the result output pin with that name.
3. If the invocation maps to a call behavior action or a call operation action with a non-empty list of argument pins and a result source element that is a result pin corresponding to a return parameter, then the action is wrapped in a structured activity node with a single output pin and an object flow from the formerly mapped result source element to the output pin. The structured activity node then becomes the effective action for the mapping and its output pin becomes the new result source element.

NOTE. Wrapping a call action in a structured activity node as above ensures that the invocation effectively produces a null token in the case that the call action itself does not actually fire because its input pin multiplicities are not satisfied.

17.8 Tuples

1. An empty tuple (i.e., a positional tuple with no argument expressions) is mapped to nothing. A non-empty tuple is mapped to a structured activity node containing the mapping of each of its argument expressions. There is a control flow from the structured activity node to the invocation action taking input from the tuple mapping.
2. For an argument for an `in` parameter, the argument expression is mapped as usual for an expression. The result source element of such an expression provides the source for setting the value of the associated parameter, unless conversion is required. If collection conversion is required, then the result source element of the argument expression is connect by an object flow to an invocation of the `Collection::toSequence` operation (see 11.7.2), and the result of that invocation provides the source for setting the value of the associated parameter, unless bit string conversion is also require. If bit string or real conversion is required, then either the result source element of the argument expression or the result of the `toSequence` invocation, if collection conversion was required, is connected by an object flow to an invocation of the `BitStringFunctions::toBitString` function (see 11.4.6) or `IntegerFunctions::ToReal` (see 11.4.2), respectively, and the result of that invocation provides the source for setting the value of the associated parameter. If the tuple is in a behavior invocation expression or sequence operation expression to which indexing from 0 applies, and the parameter is an affected `index` parameter (see 8.3.9), then the mapping of the argument expression is adjusted as for the index expression of a sequence access expression (see 17.16).
3. For an argument for an `out` parameter, the argument expression is mapped as a left hand side of an assignment (see 17.24): an argument that is a local name is mapped as a fork node while an argument that is a feature reference is mapped as write structural feature value action. The output from the invocation action for the corresponding parameter provides the assigned value.
4. For an argument for an `inout` parameter, the argument expression is mapped twice (as given above): once as for an `in` parameter, to provide the input value for the parameter, and once as for an `out` parameter, to provide the target for the output value.

17.9 Behavior Invocation Expressions

1. A behavior invocation expression whose qualified name disambiguates to a feature reference is mapped as if it were a feature invocation expression (see 17.10). Otherwise, a behavior invocation expression is mapped as either a behavior call or an association read.
2. A behavior invocation expression whose qualified name resolves to a behavior maps to a call behavior action for the named behavior.

   If the behavior invocation expression has a non-empty tuple, then the call behavior action is the target of a control flow whose source is the structured activity node mapped from the tuple.

   Each input pin of the call behavior action corresponds to an `in` or `inout` parameter of the called behavior. If there is an argument expression for that parameter in the tuple, then the input pin is the target of an object flow whose source is the result source element of the argument expression.

Similarly, each output pin of the call behavior action (other than the output pin for a `return` parameter) corresponds to an `out` or `inout` parameter. If there is an argument expression for that parameter in the tuple, then the output pin is the source of an object flow whose target is assigned value input for the argument expression.

NOTE. Call behavior action pins corresponding to unmatched parameters remain unconnected.

If the behavior does not have a `return` parameter, then the behavior invocation expression has no result source element. Otherwise, the output pin of the call behavior action corresponding to that parameter is the result source element for the behavior invocation expression, unless indexing from 0 applies to the behavior invocation expression and the invocation is of a library function whose return value is affected by this (see 8.3.9), in which case the output pin is connected by an object flow to an invocation of the `IntegerFunctions::'-'` function (see 11.4.2) whose second argument is 1, and the result of that invocation provides the result source element for the behavior invocation expression.

3. A behavior invocation expression whose qualified name resolves to an association end maps to a read link action with end data for the ends of the named association. Except for the end data for the target end, the value input pins for each end are the target of an object flow from the result source element of the mapping of the corresponding argument expression. The result output pin of the read link action is the result source element for the association selection.

17.10 Feature Invocation Expressions

1. A feature invocation expression is mapped as either a single instance feature invocation or a sequence feature invocation. For each kind of invocation, the result source element of the mapping of the feature expression is connected by an object flow to the appropriate target activity node.
2. A single instance feature invocation is mapped as either a non-destructor operation call, an explicit destructor call, an implicit destructor call or a signal send.
3. A sequence feature invocation is mapped as an expansion region similarly to a `collect` expression.

Operation Call

4. An operation call (that is not a destructor call) maps to a call operation action for the named operation. The result source element mapped from the primary expression of the feature invocation expression is connected by an object flow to the target input pin of the call operation action.
5. The call operation action has argument and result input and output pins corresponding to the parameters of the operation. These pins are connected to the appropriate mapping of argument and result expressions from the tuple (see 17.8). If the operation has a `return` parameter, then the output pin of the call operation action corresponding to that parameter is the result source element for the feature invocation action. Otherwise it has no result source element.

Destructor Call

6. If an operation call is a destructor call, and the feature invocation expression is *not* itself within the method of a destructor, then the call operation action is followed by a destroy object action for the target object with isDestroyOwnedObjects=true and isDestroyLinks=true. If the feature invocation *is* within the method of a destructor, the destroy object action is conditioned on a test that the target object is *not* the context object.

NOTE. Object destruction is always done with isDestroyOwnedObjects=true and isDestroyLinks=true, because this is the expected high-level behavior for object destruction.

7. If an operation call is an implicit object destruction expression, then it is mapped to just a destroy object action, as above, without any operation call.

Signal Send

8. A signal send maps to a send signal action for the named signal. The result source element mapped from the target expression of the feature invocation expression is connected by an object flow to the target input pin of the send signal action.

   The send signal action has argument input pins corresponding to the attributes of the signal. Each argument input pin of the send signal action is the target of an object flow whose source is the result source element of the argument expression (if there is one) mapped from the tuple (see 17.8) for the corresponding signal attribute.

   A signal send has no result source element.

17.11 Super Invocation Expressions

Once the target operation a super invocation expression is determined, the expression is mapped as a behavior invocation to the method of that operation (see 17.9).

17.12 Instance Creation Expressions

1. An instance creation expression maps as an object creation expression or a data value creation expression.

Object Creation Expression

2. An object creation expression maps as either a constructed object creation or a constructorless object creation. If the class of the object being created is an active class, then the mapping also includes the starting of the behavior of that object.

3. If the object creation expression is *not* constructorless, then the expression maps to a create object action for the class of the constructor operation. The result of the create object action is used as the target instance for an invocation of the constructor, mapped as for a feature invocation expression (see 17.10). The result source element of the object creation expression is the result output pin of the call operation action for the constructor operation.

4. If the object creation expression is constructorless, then the expression maps to a create object action for the identified class. If none of the attributes owned or inherited by the class have default values, then the result source element of the expression is the result output pin of the create object action. Otherwise, the result output pin of the create object action is connected by an object flow to a control-flow sequenced set of structured activity nodes containing write structural feature actions for setting the default values of any attributes of the newly create object that have them.

   NOTE. It is possible to notate in Alf a constructorless instance creation for any class. However, default values for attributes are value specifications (see UML Superstructure, 7.3.44), and the only kind of value specifications supported in fUML are literal specifications and instance values, not general expressions (see fUML Specification, 7.2.2.1). The mapping given here will support any kind of value specification as a default value, but the result will not conform to the fUML subset if the value specifications are outside that subset.

5. If the class of the object being created is an active class, then a fork node is added to the mapping with an object flow from the original result source element, and that fork node becomes the new result source element. The fork node is connected by object flows to the object input pins of start object behavior actions for the class of the object being created and for each direct or indirect parent class that has a classifier behavior, unless that classifier behavior is redefined in another parent class or in the class of the object being created. In this case, the entire mapping is always placed within a structured activity node.

   NOTE. Classifier behaviors with parameters are not supported by Alf, nor is the asynchronous starting of an instance of an activity with parameters. However, it is possible to notate the instantiation of an activity as a class as long as the class has no parameters, in which case the activity will, in fact, begin asynchronous execution.

Data Value Creation Expression

6. A data value creation expression maps to a value specification action with an instance value for the named data type. If the tuple for the expression is non-empty, then the value specification action is the target of a control flow whose source is the structured activity node mapped from the tuple (see 17.8). Further, the result of the value specification action is fed through a sequence of write structural feature actions with values coming from the result source elements for the argument expressions.

   If the data value creation expression has an empty tuple, then the result source element is the result pin of the value specification action. Otherwise, the result source element is the result of the sequence of write structural feature actions.

17.13 Link Operation Expressions

1. A link operation expression for the operation `createLink` maps to a create link action for the named association with isReplaceAll=false for all ends. The value input pin of the end creation data for each end of the association is the target of an object flow from the result source element of the mapping of the corresponding argument expression. If an association end is ordered, then the insertAt input pin for that end is the target of an object flow from the result source element of the mapping of the corresponding index expression (which defaults to `*` if not given explicitly). If indexing from 0 applies to the link operation expression (see 8.3.13), then the mapping of the index expression is adjusted as for the index expression of a sequence access expression (see 17.16, except that the value is first tested whether it is equal to `*` and then converted to an integer and incremented only if it is not.

2. A link operation expression for the operation `destroyLink` maps to a destroy link action for the named association. The value input pin of the end creation data for each end of the association is the target of an object flow from the result source element of the mapping of the corresponding argument expression. If an association end is unordered, the isDestroyDuplicates=true. If an association end is ordered, then isDestroyDuplicates=false and the insertAt input pin for that end is the target of an object flow from the result source element of the mapping of the corresponding index expression (which defaults to `*` if not given explicitly). If indexing from 0 applies to the link operation expression (see 8.3.13), then the mapping of the index expression is adjusted as for the index expression of a sequence access expression (see 17.16, except that the value is first tested whether it is equal to `*` and then converted to an integer and incremented only if it is not.

3. A link operation expression for the link operation `clearAssoc` maps to a clear association action for the named association. The object input pin of clear association action is the target of an object flow from the result source element of the mapping of the argument expression.

17.14 Class Extent Expressions

1. A class extent expression maps to a read extent action for the named class. The result output pin of the read extent action is the result source element for the class extent expression.

17.15 Sequence Construction Expression

Collection Object Creation Expression

1. A sequence construction expression that does not have multiplicity is mapped as an instance creation expression (see 17.12) with a constructor for the collection class given by the type name (see 11.7). The argument expression for the constructor is mapped as below for a sequence construction expression with multiplicity for the sequence type of the collection class and the sequence elements from the original expression.

Sequence Element List

2. A sequence construction expression that has multiplicity and a sequence list expression with a non-empty expression list is mapped to a structured activity node with a single output pin whose type and multiplicity, are as specified for the expression. The output pin is the result source element for the expression.
3. Each element expression is mapped inside the structured activity node, with an object flow from its result source element to the structured activity node output pin. If bit string or real conversion is required on an element, then the result source element of the element expression is connected by an object flow to an invocation of the `BitStringFunction::ToBitString` function (see 11.4.6) or `IntegerFunctions::ToReal` function (see 11.4.2), respectively, and the result of that invocation are used as the result source for the element expression. If there is more than one element expression, then the mapping for each element expression is wrapped in its own structured activity node and they are connected sequentially by control flows.

Sequence Range

4. A sequence construction expression that has multiplicity and a sequence range expression is mapped to a structured activity node with the range upper and lower expressions mapped inside it. The result source elements of the upper and lower expressions are connected by object flows to input pins of a loop node in the structured activity node. The loop node also has a third input pin that has a multiplicity lower bound of 0. The output pin corresponding to this third input pin is the result source element for the sequence range expression
5. The loop node is iterative, continually incrementing the value in its first loop variable until it reaches the value of its second loop variable. On each iteration, it appends the value of its first loop variable to the list in its third loop variable, which, at the end of the iteration, thus contains the desired sequence.

Empty Collections

6. A sequence construction expression that has multiplicity and an empty expression list maps to a value specification action for a literal null. The result output pin of the value specification has the

type given for the sequence list expression and the multiplicity `[1..1]`. It is the result source element for the expression.

7. The keyword `null` is mapped as `any[]{}`.

17.16 Sequence Access Expressions

1. A sequence access expression is mapped to a call to the primitive behavior `Alf::Library::PrimitiveBehaviors::SequenceFunctions::At` (see 11.4.7). The result source element of the primary expression of the sequence access expression is connected by an object flow to the first argument input pin of the call behavior action. The result source element of the index expression is connected by an object flow to the second argument input pin. The result output pin of the call behavior action is the result source element for the sequence access expression.
2. If indexing from 0 applies to the sequence access expression (see 8.3.16), then the mapping of its index expression is adjusted as follows. The result source element of the index expression is connected by an object flow to the first argument pin of a call to the primitive behavior `Alf::Library::PrimitiveBehaviors::IntegerFunctions::'+'` (see 11.4.2), and the result of a value specification action for the value 1 is connected to the second argument pin. The result output pin of the call behavior action is then used as the result source element for the index.

17.17 Sequence Operation Expressions

1. A sequence operation expression is mapped as a behavior invocation expression (see 17.9) for the referent behavior, with the target primary expression as the first behavior argument. The result source element for the sequence operation expression is that of the behavior invocation expression.

17.18 Sequence Reduction Expression

1. A sequence reduction expression is mapped to a reduce action with the named behavior as the reducer. The collection input pin is the target of an object flow from the result source element of the mapping of the input expression. The result output pin of the reduce action is the result source element for the reduction expression.

17.19 Sequence Expansion Expressions

1. A sequence expansion expression maps to an expansion region with a single input expansion node. Except for the `iterate` operation, the expansion region has mode=parallel. For the `iterate` operation, the expansion region has mode=iterative.
2. The input expansion node has the same type as the primary expression. It is the target of an object flow from the result source element of the mapping of the primary expression.
3. The argument expression is mapped inside the expansion region. The input expansion node is connected by an object flow to a fork node within the expansion region that acts as the assigned source for references to the expansion variable within the mapping of the argument expression.
4. The specific mapping for each kind of sequence expansion operation is further discussed in subsequent subclauses.

`select` and `reject` Expressions

5. A `select` or `reject` expression is mapped as a sequence expansion expression (see 17.19). The expansion region from this mapping has an output expansion node of the same type as the primary expression of the sequence expansion expression. This node is the result source element for the overall sequence expansion expression.
6. The result source element of the mapping of the argument expression is the source of the decision input flow for a decision node inside the expansion region. The decision node also has an incoming object flow from the expansion variable fork node and an outgoing object flow to the output expansion node. For a `select` operation, the guard on the outgoing object flow is `true`. For a reject operation, it is `false`.

`collect` and `iterate` Expressions

7. A `collect` or `iterate` expression is mapped as a sequence expansion expression (see 17.19). The expansion region has an output expansion node of the same type as the argument expression. The result source element of the mapping of the argument expression is connected by an object flow inside the expansion region to the output expansion node.
8. For an `iterate` operation, the expansion region has mode=iterative. Otherwise it has the normal mode=parallel.

`forAll`, `exists` and `one` Expressions

9. A `forall` expression is mapped the same as a `reject` expression, except that the output expansion node of the expansion region is connected by an object flow to a call behavior action for the library `isEmpty` function. The result output pin of the call behavior action is the result source element for the `forAll` expression.
10. An `exists` expression is mapped the same as a `select` expression, with the addition that, inside the expansion region, the decision node is not directly connected to the output expansion node but, rather, is connected to a fork node that is connected both to the output expansion node and to an activity final node. Further, the output expansion node of the expansion region is connected by an object flow to a call behavior action for the library `notEmpty` function. The result output pin of the call behavior action is the result source element for the `exists` expression.

    NOTE. The inclusion of an activity final node within the expansion region is intended to terminate the region as soon as an element is found for which the `Boolean` expression is `true`. (Despite its name, an activity final node within a structured node such as an expansion region only terminates that structured node.)
11. A `one` expression is mapped the same as a `select` expression, except that the output expansion node of the expansion region is connected by an object flow to a call behavior action for the library `size` function. The result output pin of the call behavior action is then connected by an object flow to the input pin of a test identity action whose other input pin is connected to a value specification action for the value 1. The result output pin of the test identity action is the result source element of the `one` expression.

`isUnique` Expressions

12. An `isUnique` expression is mapped as a `collect` expression. The expansion output node of the expansion region mapped from the collect expression is connected by an object flow to a fork not which is then connect by object flows to an input expansion node and an input pin of another

expansion region. The second expansion region is mapped similarly to a `forAll` expression, with the condition that the `count` of each value its sequence is 1. The result source element for the `isUnique` expression is the result output pin of the `isEmpty` call on the output of the second expansion region.

17.20 Increment and Decrement Expressions

1. An increment or decrement expression is mapped to a call behavior action for the + function (for increment) or the – function (for decrement) from the library package `Alf::Library::PrimitiveBehaviors::IntegerFunctions` (see 11.4.2).. The second argument input pin of the call behavior action is connected by an object flow to the result output pin of a value specification action for the value 1. The result output pin of the call behavior action is connected by an object flow to a fork node, which acts as the source element when the expression is an assigned source.
2. The operand is mapped first considered as an effective argument expression. If the increment or decrement expression is a prefix expression, then the result source element of this mapping is connected by an object flow to the first argument input pin of the call behavior action and the assigned source element is also the result element for the expression. If it is a postfix expression, then the result source element of the operand expression mapping is first connected to a fork node and then that is connected to the argument input pin, and the fork node is the result source element for the expression.
3. The operand is also mapped as a left-hand side to which is assigned the result of the call behavior action (see 17.24).
4. If the operand has an index, then the index expression is only mapped once. The result source element of the index expression is connected by an object flow to a fork node, which is used as the source for the index value in the mapping of the operand expression both as an argument and as the left hand side of the assignment.

17.21 Unary Expressions

Boolean Unary Expressions

1. A Boolean unary expression with a Boolean negation operator is mapped as the equivalent behavior invocation (see 17.9)for the function `Alf::Library::PrimitiveBehaviors::BooleanFunctions::'!'` (see 11.4.1) on the operand expression.

BitString Unary Expressions

2. A BitString unary expression with a BitString negation operator is mapped as the equivalent behavior invocation (see 17.9) for the function `Alf::Library::PrimitiveBehaviors::BitStringFunctions::'~'` (see 11.4.6) on the operand expression. Note that this includes the possibility of bit string conversion on the operand expression.

Numeric Unary Expressions

3. A numeric unary expression with a plus operator is mapped as its operand expression. A numeric unary expression with a minus operator is mapped as the equivalent behavior invocation (see 17.9) for the function `Alf::Library::PrimitiveBehaviors::IntegerFunctions::Neg` (see 11.4.2) `Alf::Library::PrimitiveBehaviors::RealFunctions::Neg` (see 11.4.3) on the operand expression.

Cast Expressions

4. If the named type is `any` or is a supertype of the type of the operand expression, then a cast expression is mapped as its operand expression, except that, if the result source element is a typed element, it is typed with the type of the cast expression, rather than the type of the argument expression.
5. If the named type is a classifier, then a cast expression is mapped like a `select` expression (see 17.19) whose condition is a read is classified object action for the type of the cast expression ("`instanceof`" operator).
6. Otherwise, a cast expression is mapped like a `select` expression (see 17.19) whose condition at least includes a read is classifier object action ("`instanceof`" operator) for the type of the cast expression.

   If the type of the cast expression is `Integer`, `UnlimitedNatural` or `BitString`, then the condition also includes additional tests to allow casting between `Integer` and `UnlimitedNatural` and between `Integer` and `BitString`. In the later cases, value conversions are performed by inserting a behavior invocation (see 17.9) of the appropriate primitive behavior from the `Alf::Library::PrimitiveBehaviors` package (see 11.4), as given in Table 17.1.

Table 17.1 Primitive Behaviors Used for Conversions

Conversion	Behavior
`Integer` to `UnlimitedNatural`	`IntegerFunctions::ToUnlimitedNatural`
`UnlimitedNatural` to `Integer`	`UnlimitedNaturalFunctions::ToInteger`
`Integer` to `BitString`	`BitStringFunctions::ToBitString`
`BitString` to `Integer`	`BitStringFunctions::ToInteger`
`Integer` to `Real`	`IntegerFunctions::ToReal`
`Real` to `Integer`	`RealFunctions::ToInteger`

Isolation Expressions

7. An isolation expression is mapped as a structured activity node with isIsolated=true and the operand expression mapped inside it. The structured activity node has a single output pin with the type and multiplicity of the operand expression. The result source element from the mapping of the operand expression is connected inside the structured activity node by an object flow to the output pin. The result source element for the isolation expression is the output pin of the structure activity node.

17.22 Binary Expression

Arithmetic Expressions

1. An arithmetic expression is mapped as a behavior invocation (see 17.9) for the corresponding primitive behavior from the `Alf::Library::PrimitiveBehaviors` package (see 11.4), as given in Table 17.2. Note that this includes the possibility of performing real conversion on either operand, as necessary.

Table 17.2 Primitive Behavior Equivalents for Arithmetic Operators

Operator	`Integer` Behavior (isConcatenation=false, isReal=false)	`Real` Behavior (isReal=true)	`String` Behavior (isConcatenation=true)
+	`IntegerFunctions::'+'`	`RealFunctions::'+'`	`StringFunctions::'+'`
-	`IntegerFunctions::'-'`	`RealFunctions::'-'`	
*	`IntegerFunctions::'*'`	`RealFunctions::'*'`	
/	`IntegerFunctions::'/'`	`RealFunctions::'/'`	
%	`IntegerFunctions::'%'`		

Shift Expressions

2. A shift expression is mapped as a behavior invocation (see 17.9) for the corresponding primitive behavior from the `Alf::Library::PrimitiveBehaviors` package (see 11.4), as given in Table 17.3. Note that this includes the possibility of performing bit string conversion on the first operand.

Table 17.3 Primitive Behavior Equivalents for Arithmetic Operators

Operator	Behavior
<<	`BitStringFunctions::'<<'`
>>	`BitStringFunctions::'>>'`
>>>	`BitStringFunctions::'>>>'`

Relational Expressions

3. A relational expression is mapped as a behavior invocation (see 17.9) for the corresponding primitive behavior from the `Alf::Library::PrimitiveBehaviors` package (see 11.4), as given in Table 17.4. Note that this includes the possibility of performing real conversion on either operand, as necessary.

Table 17.4 Primitive Behavior Equivalents for Relational Operators

Operator	`Integer` Behavior (isUnlimitedNatural=false, isReal=false)	`Real` Behavior (isReal=true)	`UnlimitedNatural` Behavior (isUnlimitedNatural=true)
<	`IntegerFunctions::'<'`	`RealFunctions::'<'`	`UnlimitedNaturalFunctions::'<'`
>	`IntegerFunctions::'>'`	`RealFunctions::'>'`	`UnlimitedNaturalFunctions::'>'`
<=	`IntegerFunctions::'<='`	`RealFunctions::'<='`	`UnlimitedNaturalFunctions::'<='`
>=	`IntegerFunctions::'>='`	`RealFunctions::'>='`	`UnlimitedNaturalFunctions::'>='`

Logical Expressions

4. A logical expression is mapped as a behavior invocation (see 17.9) for the corresponding primitive behavior from the `Alf::Library::PrimitiveBehaviors` package (see 11.4), as given in Table 17.5. Note that this includes the possibility of applying bit string conversion to one or both operands, if the operator is bit-wise.

Table 17.5 Primitive Behavior Equivalents for Logical Operators

Operator	`Boolean` Behavior (isBitWise=false)	`BitString` Behavior (isBitWise=true)
`&`	`BooleanFunctions::'&'`	`BooleanFunctions::'&'`
`^`	`BooleanFunctions::'^'`	`BooleanFunctions::'^'`
`|`	`BooleanFunctions::'|'`	`BooleanFunctions::'|'`

Classification Expressions

5. A classification expression maps to a read is classified object action for the named classifier with. If the classification operator is `instanceof`, then isDirect=false. If the operator is `hasType`, then isDirect=true. The object input pin of the action is the target of an object flow from the result source element for the mapping of the operand expression. The result output pin of the action is the result source element for the classification expression.

Equality Expressions

6. An equality expression is mapped to a test identity action. If the expression uses the operator `==`, and both operand expressions have a multiplicity lower bound of 1, then the input pins of the action are the targets of object flows from the result source elements for the mappings of the argument expressions. The output pin of the action is the result source pin for the equality expression.
7. If either operand expression has a multiplicity lower bound of 0, then the result of that expression is first tested for being not empty using the library function `Alf::Library::PrimitiveBehaviors::SequenceFunctions::NotEmpty` (see 11.4.7). The test identity action is executed only if both argument expressions are non-empty. Otherwise, the equality expression is true only if both argument expressions are empty.

   NOTE. Despite the extra checks described above, the mapping for an equality expression still always evaluates the operand expressions exactly *once.*
8. If real conversion is required on an operand, then the result source element of the operand expression is connected by an object flow to an invocation of the `IntegerFunctions::ToReal` function (see 11.4.2), and the result of that invocation is used as the result source for the operand expression in the mapping above.
9. An equality expression that uses the operator `!=` is mapped as above, but the result output pin of the test identity action is connected by an object flow to the argument input pin of a call behavior action for the library function `Alf::Library::PrimitiveBehaviors:: BooleanFunctions::'!'` (see 11.4.1). The result source element is the result output pin of the call behavior action.

Conditional Logical Expressions

10. A conditional-and expression is mapped like a conditional-test expression (see 17.23) whose first two operand expressions are the same as those of the conditional-and expression and whose third operand expression is `false`.
11. A conditional-or operator expression is mapped like a conditional-test expression (see 17.23) whose first and third operand expressions are the same as the two operand expressions of the conditional-or expression and whose second operand expression is `true`.

12. A null-coalescing expression is mapped like a conditional-test expression (see 17.23) whose first operand expression is an invocation of the library function `Alf::Library::PrimitiveBehaviors::SequenceFunctions::NotEmpty` with an argument that is the assignment of the result of the first operand expression of the null-coalescing expression to a temporary local name (not otherwise being used), whose second operand expression is a reference to the temporary local name and whose third operand expression is the second operand expression of the null-coalescing expression.

17.23 Conditional-Test Expressions

1. A conditional-test expression maps to a decision node with an incoming control flow from an initial node. The decision input flow for the decision node has as its source the result source element from the mapping of the first operand expression.

   The decision node has two outgoing control flows with the guards `true` and `false`. The `true` flow has as its target a structured activity node that contains the mapping of the second operand expression. The `false` flow has as its target a structured activity node that contains the mapping of the third operand expression.

   The result source elements from the mapping of the second and third operand expressions are connected by object flows to a merge node (outside either structured activity node). This merge node is the result source element for the conditional-test expression.

2. For any name assigned in either (or both) of the second and third operand expressions, an output pin is added to the structured activity nodes for both the second and third operand expressions. Within each structured activity node, if the name is assigned in the corresponding operand expression, then the assigned source for the name after the operand expression is connected to the output pin. If the name is not assigned in the corresponding operand expression, but has an assigned source before the operand expression, then an additional structured activity node is added to the mapping of the operand expression as follows.

   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
   - There is an object flow from the assigned source for the name before the operand expression to the input pin of the structured activity node.

   The output pin of the added structured activity node is then connected by an object flow to the output pin corresponding to the name on the enclosing structured activity node for the argument expression.

   Each pair of output pins on the structured activity nodes for the operand expressions corresponding to the same name are connected by object flows to a merge node, which is connected by an object flow to a fork node. The fork node is the source for the assigned value of the name after the conditional-test expression.

17.24 Assignment Expressions

1. The mapping of an assignment expression depends on whether it is a simple or compound assignment and, if it is a simple assignment whether it has a name or feature left-hand side and whether or not it has an index.

2. As an assigned source, an assignment expression maps to the result source of the expression.
3. If no conversion is required then the result source element of the right-hand side of an assignment expression, as referenced below, should be considered to the the result source element of the mapping of the right-hand side expression. If collection conversion is required, then the result source element of the argument expression is connect by an object flow to an invocation of the `Collection::toSequence` operation (see 11.7.2), and the result of that invocation acts as the result source element for the right-hand side, unless bit string conversion is also require. If bit string conversion is required, then either the result source element of the argument expression or the result of the `toSequence` invocation, if collection conversion was required, is connected by an object flow to an invocation of the `BitStringFunctions::toBitString` function (see 11.4.6) or `IntegerFunctions::ToReal` (see 11.4.2), respectively, and the result of that invocation acts as the result source element for the right-hand side.

Simple Assignment: Name Left-Hand Side, without Index

4. If the left-hand side is a name without an index, then a simple assignment maps to a fork node. The result source element from the mapping of the right-hand side is connected to the fork node by an object flow. The fork node is the result source element for the assignment expression and also the source for the assigned value for the name.

Simple Assignment: Name Left-Hand Side, with Index

5. If the left-hand side is a name with an index, then a simple assignment maps to a call behavior action for the library behavior `Alf::Library::SequenceFunctions::ReplaceAt` (see 11.4.7). The assigned source for the name from the left-hand side is connected b an object flow to the `seq` argument input pin of the call behavior action. The result source element from the mapping of the right-hand side is connected to the `element` argument input pin and the result source element from the mapping of the index expression is connected to the `index` argument input pin (if indexing from 0 applies to the left-hand side – see 8.8– then the mapping of the index expression is adjusted as for the index expression of a sequence access expression – see 17.16). The `seq` output pin of the call behavior action is connected by an object flow to a fork node, which is the result source element for the assignment expression and also the source for the assigned value for the name after the expression.

Simple Assignment: Feature Left-Hand Side, without Index

6. If the left-hand side is a property reference but has no index, then the mapping of a simple assignment depends on the multiplicity upper bound of the right-hand side expression.
7. If the right-hand side expression has a multiplicity upper bound of 0, then the simple assignment maps to a clear structural feature action for the identified property. If the right-hand side expression is a sequence construction expression for an empty set, then it is not mapped at all. Otherwise, the right-hand side expression is mapped inside a structured activity node, with a control flow from the structured activity node to the clear structural feature action. There is no result source element for the assignment.
8. If the right-hand side expression has a multiplicity upper bound of 1, then the simple assignment maps to an add structural feature value action for the identified property with isReplaceAll=true. The result source element from the mapping of the right-hand side expression is connect by an object flow to a fork node that has a further object flow to the value input pin of the add structural feature value action. The fork node is the result source element for the assignment.

9. Otherwise, the simple assignment maps to a clear structural feature value action for the identified property followed by an expansion region. The result source element from the mapping of the right-hand side expression is connected by an object flow to a fork node that has a further object flow to an input expansion node of the expansion region. The expansion region contains an add structural feature value action for the property with isReplaceAll=false and an incoming object flow from the input expansion node to its value input pin. If the property is ordered, then the insertAt input pin has an incoming object flow from a value specification action for the unbounded value `*`. The fork node is the result source element for the assignment.

Simple Assignment: Feature Left-Hand Side, with Index

10. If the left-hand side has an index, then the mapping of a simple assignment includes a structured activity node containing the mapping of the index expression (if indexing from 0 applies to the left-hand side – see 8.8– then the mapping of the index expression is adjusted as for the index expression of a sequence access expression – see 17.16). The further mapping of the assignment expression then depends on the multiplicity upper bound of the right-hand side expression.

11. If the right-hand side expression has a multiplicity upper bound of 0, then the simple assignment maps to a remove structural feature value action for the identified property with isRemoveDuplicates=false and an incoming object flow into its removeAt input pin from the result source element from the mapping of the index expression. If the right-hand side expression is a sequence construction expression for an empty collection, then it is not mapped at all. Otherwise, the right-hand side expression is mapped inside a structured activity node, with a control flow from that structured activity node to the structured activity node containing the mapping of the index expression. There is no result source element for the assignment.

12. If the right-hand side expression has a multiplicity upper bound of 1, then the simple assignment maps to a remove structural feature value action for the identified property with isRemoveDuplicates=false followed by an add structural feature value action with isReplaceAll=false. The result source element of the mapping of the index expression is connected by an object flow to a fork node, which then has object flows to the removeAt input pin of the remove structural feature value action and the insertAt input pin of the add structural feature value action. The right-hand side expression is mapped inside a structured activity node, which is connected by a control flow to the structured activity node for the index expression. The result source element of the mapping of the right-hand side expression is connected by an object flow to the value input pin of the add structural feature value action. The fork node is the result source element for the assignment.

13. If the left-hand side is a data value attribute update, then a fork node is added to the mapping for the assignment expression to be used as the source element for the assigned value of the name. The fork node is the target of an object flow whose source is determined as follows.
    - If the mapping includes a remove structural feature action, but no add structural feature action, then the result output pin of the remove structural feature action is used.
    - If the mapping includes an add structural feature action not in an expansion region, then the result output pin of the add structural feature action is used.
    - If the mapping has an add structural feature action in an expansion region, then an output expansion node is added to the expansion region and the result output pin of the add structural feature action is connected to the output expansion node by an object flow. The output expansion

node is then connected by an object flow to a mapping of the expression `ListGet(x,ListSize(x))`, where *x* represents the object flow from the expansion node.

Compound Assignment

14. A compound assignment is mapped like a simple assignment expression for which the assigned value is the result of a call behavior action for the primitive behavior corresponding to the compound assignment operator. The arguments to the call behavior action come from the result source elements of the mapping of the effective expression for the left-hand side and the right-hand side expression. However, if the left-hand side is a property reference, then the primary expression for the reference and any index expression are only mapped once with their values used both in the mapping of the effective expression for the left-hand side and the updating of the left-hand side as a structural feature.

18 Statements Mapping

18.1 General

This clause defines the mapping of Alf statements to UML. The abstract syntax for Alf statements is described in Clause 14.

1. Every statement is mapped to a single activity node (which may be a structured activity node with nested structure). The specific mapping for each kind of statement is described in the following subclauses.
2. If the static analysis of assignment indicates that a name with an assigned source has a different assigned source after a statement than before the statement, and the statement maps to a structured activity node (but not a conditional node or a loop node), then an input pin corresponding to that name is added to the structured activity node. This input pin is the target of an incoming object flow from the assigned source of the name before the statement. The input pin is also connected by an outgoing object flow to a fork node contained in the structured activity node. This fork node acts as the assigned source for the name within the mapping of the statement.
3. A block maps to the union of the nodes mapped from the statements in it. In addition, unless the block is in a block statement annotated as being parallel (see 18.3), the node mapped from each statement other than the last has a control flow targeted to the node mapped from the next statement.
4. A statement for which `@isolated` is allowed is always mapped to structured activity node, and annotating it `@isolated` results in the isIsolated property on this node being `true`. (Other annotations are discussed with the description of the statements to which they apply.)

18.2 In-Line Statements

1. An in-line statement maps to an opaque action with a body given by the code text within the statement with a corresponding language string given by the language name in the statement. (The language string is given by the actual name and so does not include the single quotes that appear in the representation of an unrestricted name.)
2. Depending on the mechanism used by a specific implementation, the opaque action may also have input and/or output pins with connected object flows for providing data to and from the code executed by the opaque action. However, the details of such mapping are not defined by the Alf standard.

   NOTE. The fUML subset does not include opaque actions, so an activity mapped from Alf including an in-line statement will not be executable as fUML. The execution behavior of such an activity is implementation specific.

18.3 Block Statements

1. A block statement maps to a structured activity node containing all the activity nodes and edges mapped from its block. If the block statement is not parallel, then the nodes mapped from the statements of the block have control flows between them enforcing their sequential execution. If the block statement is parallel, then there are no such control flows.

18.4 Empty Statements

1. An empty statement maps to an empty structured activity node.

   NOTE. Mapping an empty statement to *something* preserves the rule that every statement maps to a single activity node and allows for the general rule for the mapping of control flow within a statement sequence.

18.5 Local Name Definition Statements

1. A local name declaration statement is mapped as if it was an expression statement (see 18.6) with an assignment expression (see 17.24) having the local name as its left-hand side and the expression as its right-hand side.

18.6 Expression Statements

1. An expression statement maps to a structured activity node containing the activity nodes and edges mapped from its expression (see Clause 17).

18.7 `if` Statements

Clauses

1. An `if` statement maps to a conditional node. Each `if` clause maps to a clause of the conditional node. For a final `if` clause, the test part of the clause is a single value specification action for a Boolean literal with value "`true`".
2. Each clause specified in a concurrent clause set has as predecessors all clauses specified by the immediately preceding concurrent clause set (if any) in the sequential clause set for the conditional node. A final clause has as its predecessor all clauses specified by the immediately preceding concurrent clause set.
3. The isAssured and/or isDetermined properties of the conditional node are set according to whether the if statement is assured or determined.

Output Pins

4. The conditional node has a result output pin corresponding to each local name that is assigned in *any* of the `if` clauses. Therefore, each clause of the conditional node also must have a body output pin from within the clause identified for each result pin of the conditional node. If a newly defined name corresponding to a conditional-node result pin is unassigned before a clause and not assigned within the clause, then a value specification action with a literal null value is added to the mapping of the clause, and the result pin of that action is used as the clause body output pin corresponding to the local name. If a name is assigned within a clause and the assigned source for that name within the clause is a pin on an action within the body of the clause, then that pin is used as the clause body output pin corresponding to that local name. Otherwise, a structured activity node is added to the mapping of the clause as follows.
   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
   - There is an object flow from the assigned source for the name after the clause (which may be from inside or outside the clause) to the input pin of the structured activity node.

The output pin of the structured activity node is then used as the clause body output pin corresponding to the name.

18.8 `switch` Statements

Clauses

1. A `switch` statement maps to a structured activity node that contains a conditional node and the mapping for the switch expression. The switch clauses map to concurrent clauses of the conditional node. Each clause tests whether the result of the switch expression equals the result of one of the case expressions.
2. A switch default clause is mapped to a conditional node clause with a condition of `true` and with all other clauses as predecessor clauses.
3. The isAssured and/or isDetermined properties of the conditional node are set according to whether the switch statement is assured or determined..

Output Pins

4. The result and clause body output pins of the conditional node are mapped as for an `if` statement (see 18.7).

18.9 `while` Statements

Loop Node

1. A `while` statement maps to a loop node with isTestedFirst=true (see fUML Specification, 7.4.4). The loop node contains the activity nodes and edges mapped from both the condition expression (see Clause 17) and the block of the `while` statement. All the actions from the mapping of the condition expression constitute the test part of the loop node. All the actions from the mapping of the block form the body part.
2. If the result source element from the mapping of the condition expression is an output pin, then this is the decider pin for the loop node. Otherwise, a structured activity node is added inside the loop node as follows.
   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
   - There is an object flow from the result source element from the mapping of the expression to the input pin of the structured activity node.

   The output pin of the structured activity node is then used as the decider pin.

Loop Variables

3. Any name that is assigned in the condition expression or block of the `while` statement is mapped to a loop variable of the loop node. The loop variable corresponding to a name is used as the assigned source before the condition expression when mapping the `while` statement. If the name is assigned before the `while` statement, then the corresponding loop variable input pin is the target of an incoming object flow from the assigned source for the name before the `while` statement. Otherwise the loop variable input pin is unconnected.

4. If the assigned source for the name after the block of the `while` statement is an output pin, then this output pin is identified as the body output pin corresponding to the loop variable for the name. Otherwise, a structured activity node is added to the mapping of the body of the loop as follows.
   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
   - There is an object flow from the assigned source for the name after the block to the input pin of the structured activity node.

   The output pin of the structured activity node is then used as the body output pin corresponding to the name.
5. If the assigned source of a name after a `while` statement is the statement, then the source for its assigned value is the result output pin of the loop node corresponding to the loop variable for the name.

18.10 `do` Statements

Loop Node

1. A `do` statement maps to a loop node with isTestedFirst=false (see fUML Specification, 7.4.4). The loop node contains the activity nodes and edges mapped from both the block and the condition expression (see Clause 17) of the `do` statement. All the actions from the mapping of the condition expression constitute the test part of the loop node. All the actions from the mapping of the block form the body part.
2. If the result source element from the mapping of the condition expression is an output pin, then this is the decider pin for the loop node. Otherwise, a structured activity node is added inside the loop node as follows.
   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
   - There is an object flow from the result source element from the mapping of the expression to the input pin of the structured activity node.

   The output pin of the structured activity node is then used as the decider pin.

Loop Variables

3. Any name that is assigned in the condition expression or block of the `do` statement is mapped to a loop variable of the loop node. The loop variable corresponding to a name is used as the assigned source before the block when mapping the `do` statement. If the name is assigned before the `do` statement, then the corresponding loop variable input pin is the target of an incoming object flow from the assigned source for the name before the `do` statement. Otherwise the loop variable input pin is unconnected.
4. If the assigned source for the name after the condition expression of the `do` statement is an output pin, then this output pin is identified as the body output pin corresponding to the loop variable for the name. Otherwise, a structured activity node is added to the mapping of the body of the loop as follows.
   - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.

- There is an object flow from the assigned source for the name after the block to the input pin of the structured activity node.

The output pin of the structured activity node is then used as the body output pin corresponding to the name.

5. If the assigned source of a name after a `do` statement is the statement, then the source for its assigned value is the result output pin of the loop node corresponding to the loop variable for the name.

18.11 `for` Statements

Iterative `for` Statements

1. An iterative `for` statement is generally mapped to a structured activity node containing the mapping of the loop variable expressions and either a loop node or an expansion region.
2. An iterative `for` statement that does not make within it any re-assignments to names defined outside of it is mapped to an expansion region with mode=iterative (see fUML Specification, 7.4.5). The result source element from the mapping of the loop variable expressions are connected by object flows to input expansion nodes on the expansion region.
3. Otherwise, an iterative `for` statement is mapped to a loop node (see fUML Specification, 7.4.4). A `for` statement of the form

```
for (v1 in expr1, v2 in expr2,…) { stmts }
```

is mapped equivalently to

```
{
  list1 = (T[]) expr1;
  list2 = (T[]) expr2;
  …
  size = list1->size();
  i = 1;
  while (i <= size) {
    v1 = list1[i];
    v2 = list2[i];
    …
    stmts
    i++;
  }
}
```

where *list, size* and *i* are arbitrary local names not otherwise used in the enclosing activity and *T* is the type of *expr1*.

Parallel `for` Statements

4. A parallel `for` statement is always mapped to an expansion region, but with mode=parallel. Any name listed in the `@parallel` annotation for the statement is mapped to an output expansion node with that name and its assigned type. This expansion node provides the source element for the assigned value for the name after the `for` statement.

18.12 `break` Statements

1. A `break` statement maps to an empty structured activity node with a control flow to an activity final node. The activity final node is placed in the outermost structured activity node mapped from the target statement of the `break` statement.

18.13 `return` Statements

1. A return statement is mapped to a structured activity node. The structured activity node is connected by a control flow to an activity final node placed directly at the top level within the activity enclosing the return statement.
2. If the return statement has an expression, then the mapping for that expression is contained in the structured activity node for the return statement. The result source element of the mapping of the expression is connected by an object flow to an output pin of the structured activity node. This output pin is, in turn, connected by an object flow to the output activity parameter node for the return parameter of the enclosing activity.

18.14 `accept` Statements

Simple `accept` Statements

1. A simple `accept` statement maps to an accept event action (see fUML Specification, 7.5.4.2.2) with triggers for each of the named signals. If the accept statement defines a local name, then the result output pin of the accept event action is connected by an object flow to a fork node, which acts as the assigned source for the local name. Otherwise the accept event action has no result output pin.

Compound `accept` Statements

2. A compound `accept` statement maps to a structured activity node containing an accept action with triggers for each of the signals mentioned in any of the clauses of the `accept` statement. The result output pin of the accept event action is connected by an object flow to a fork node, which is further connected to conditional logic that tests the type of the received signal.
3. Each block in a compound accept statement is mapped as a block statement (see 18.3), with a control flow from the conditional logic mentioned above, corresponding to the appropriate type(s) of signals for that block. The fork node mentioned above acts as the source for the assigned value of the local name defined in the accept clause for the block, if any.
4. For any name assigned within one or more blocks of the `accept` statement, a corresponding output pin is added to the structured activity node mapped from the `accept` statement. This output pin becomes the source for the assigned value for the name after the `accept` statement. The assigned source from each block assigning the name is connected by an object flow to the corresponding output pin. For each block that does not assign the name, if the name was unassigned before the block, then a value specification action with a literal null value is added to the mapping of the block, and the result pin of that action is connected by an object flow to the corresponding output pin for the name. Otherwise, a structured activity node is added to the mapping of the block as follows.
    - The structured activity node has one input pin and one output pin, with an object flow from the input pin to the output pin contained within the structured activity node.
    - There is an object flow from the assigned source of the name to the input pin of the structured activity node.

- There is an object flow from the output pin of the structured activity node to the corresponding output pin for the name of the enclosing structured activity node for the `accept` statement.

18.15 `classify` Statements

1. A `classify` statement maps to a structured activity node containing a reclassify object action (see UML Superstructure, 11.3.9 and fUML Specification, 7.5.4.2.5) and the mapping of the target object expression, the result source element of which is connected by an object flow to the object input pin of the reclassify object action. The from classes for the `classify` statement are the old classifiers for the reclassify object action and the to classes are the new classifiers. If the `classify` statement is reclassify all, then the reclassify object action has isReplaceAll=true.
2. If any of the to classes of the `classify` statement are active classes with classifier behaviors, then a fork node is added to the mapping (inside the structure activity node) with an object flow from the original result source element. The fork node is then used as the source for the object input pin of the reclassify object action, as well as the source for the object input pins of start object behavior actions for each of the to class with classifier behaviors. Control flows are also added from the reclassify object action to each of the start object behavior actions.

   NOTE. According to the fUML semantics of start object behavior actions (see fUML Specification, 8.6.4.2.10, 8.3.2.2.19, and 8.4.3.2.10), if a classifier behavior is already running for an object, than a start object behavior action for that classifier behavior has no effect for that object.

19 Units Mapping

19.1 General

This clause defines the mapping of Alf units to UML. The abstract syntax for Alf statements is described in Clause 14.2.

Unit Definition

1. A unit definition maps to a specific kind of namespace according to the namespace definition for the unit definition, as given in the appropriate subsequent subclause. If a namespace declaration is given, the namespace mapped from the unit is an owned member of the declared namespace. If no namespace declaration is given, then the unit must be a model unit and what namespace owns it, if any, is not defined by the Alf specification.
2. If the unit is a model unit, then it has empty visibility. Otherwise, the unit visibility is given by the stub declaration for it in the definition of its owning namespace.

Import Reference

3. An element import reference maps to an element import from the namespace to the named imported element. The element import visibility is as given by the import visibility indicator. If there is an alias part, then the given unqualified name becomes the element import alias.
4. A package import reference maps to a package import from the namespace to the named package. The package import visibility is as given by the import visibility indicator.

Stereotype Annotation

5. A stereotype annotation, other than for the special cases given in Table 10.1, maps formally to the application of the identified stereotype to the element mapped from the annotated member.. However, an implementation may also use such stereotypes to specify special implementation-specific semantics for the annotated element, except for the standard stereotypes «Create» and «Destroy», which are used in the standard Alf mapping for constructors and destructors and «ModelLibrary», which is used to suppress the inclusion of implicit imports.

19.2 Namespace Definitions

1. A namespace definition maps to a namespace and its owned members, as specified for each kind of namespace definition in the appropriate subsequent subclause.
2. A visibility indicator maps to the visibility of the named element mapped from the definition containing the visibility indicator, with an empty visibility indicator mapping to a visibility kind of “package”.

19.3 Package Definitions

1. A package definition maps to a package. If the package definition is a stub, then it is mapped according to the associated subunit definition.
2. The applied profiles of a package definition map to profile application relationships from the package to each of the applied profiles.

3. Each package member is mapped according to its kind. The resulting elements are a packaged elements of the package.

See also the mapping for visibility in 19.2.

19.4 Classifier Definitions

1. A classifier definition (other than a classifier template parameter) maps to a classifier and its features, as specified for each kind of classifier in the appropriate subsequent subclause. If the classifier definition is a stub, then it is mapped according to its associated subunit definition.

Specialization

2. If the classifier definition has specialization referents, then the classifier being defined has generalization relationships with each of the referents. If the classifier definition is abstract, then the classifier has isAbstract=true. Otherwise isAbstract=false.

Template Parameters

3. If a classifier definition has owned members that are classifier template parameters, then these map classifier template parameters of the classifier. If a classifier template parameter has a specialization referent, this maps to a constraining classifier for the parameter.
4. Each template parameter has a corresponding parametered element which is a private member owned by the classifier being defined with the same name as the template parameter. If the template parameter has a constraining classifier, then the parametered element is a classifier of the same kind as the constraining classifier and it has a generalization relationship to the constraining classifier. Otherwise it is a data type. In all cases, the parametered element is abstract.

19.5 Class Definitions

1. A class definition maps to a class with isActive=false. (For active classes, see 19.6).

Class Members

2. A nested classifier definition maps to a classifier as specified in the subclause for the appropriate kind of classifier. If the nested classifier definition is a stub declaration, then the stub declaration is mapped according to the associated subunit definition. The resulting classifier is a nested classifier of the class.
3. A property definition maps to an owned attribute of the class as specified in 19.14.
4. An operation definition maps to an owned operation of the class as specified in 19.15.

See also the mapping for visibility in 19.2.

Default Constructors and Destructors

1. If a class definition has no operation definitions that are constructors, a public, concrete owned operation is added to the class with the same name as the class and no parameters and the standard «`Create`» stereotype applied. It has a corresponding method activity that is a private owned behavior of the class with the default behavior described in 10.5.3.1. Within this behavior, initializers for attributes of the class are mapped as sequenced structured activity nodes containing the mappings of the initializer expressions (see 17.24).
2. If a class definition has no operation definitions that are destructors, a public, concrete owned operation with the name "`destroy`", no parameters and the standard «`Destroy`» stereotype applied is

added to the class. It has a corresponding method activity that is a private owned behavior of the class with the default behavior described in 10.5.3.2.

19.6 Active Class Definitions

1. An active class is mapped like a passive class (see 19.4), except an active class has isActive=true and the following additional rules for mapping the classifier behavior and receptions.
2. If the behavior clause for an active class is a name, then the classifier behavior for the class is the named activity. If the behavior clause is a block, then the classifier behavior is an activity with the activity nodes and edges mapped from the block (see 18.1).
3. An active feature definition maps to an owned reception of the class as specified in 19.16. An active feature stub declaration is mapped according to the associated subunit definition.

19.7 Data Type Definitions

1. A data type definition that is not primitive maps to a data type (that is not an enumeration or a primitive type).
2. A data type definition that is primitive maps to a primitive type. This primitive type is registered as a built-in type with the execution factory at the execution locus for the unit (see 8.2 of the fUML Specification).

Data Type Members

3. A property definition maps to an owned attribute of the data type as specified in 19.14.

See also the mapping for visibility in 19.2.

19.8 Association Definitions

1. An association definition maps to an association.

Association Members

2. A property definition maps to an owned end of the association as specified in 19.14. All ends are navigable owned ends of the association.

See also the mapping for visibility in 19.2.

19.9 Enumeration Definitions

1. An enumeration definition maps to an enumeration.
2. An enumeration literal name maps to an enumeration literal that is an owned literal of the enumeration and has the given unqualified name.

19.10 Signal (and Signal Reception) Definitions

1. A signal definition maps to a signal.

Signal Members

2. A property definition maps to an owned attribute of the signal as specified in 19.14.

See also the mapping for visibility in 19.2.

Signal Reception Definition

3. A signal reception definition maps to a signal *and* a reception for the signal. The signal is mapped as if the signal reception definition was a signal definition and the signal becomes a nested classifier of the class mapped from the class definition that is the namespace of the signal reception definition. The reception becomes an owned reception of the same class.

19.11 Activity Definitions

1. An activity definition that is not primitive maps to an activity. If the activity is a classifier behavior, it is mapped as active (isActive=true). Otherwise, it is mapped as passive (isActive=false). The body of an activity maps as a block (see 18.1).
2. An activity definition that is primitive maps to an opaque behavior. An execution prototype for the opaque behavior is registered as a primitive behavior with the execution factory at the execution locus for the unit (see 8.2 of the fUML Specification). However, how this execution prototype is created is tool specific and not defined in the Alf standard.

Activity Members (Formal Parameters)

3. A formal parameter maps to an owned parameter of the activity (see 19.13). The type and multiplicity of a formal parameter are mapped as specified in 19.13.
4. Each `in` and `inout` parameter of the activity maps to an input activity parameter node for the parameter. Each such node is connected by an outgoing object flow to a fork node. This fork node acts as the (initial) assigned source for the values of the parameter within the activity (see also 17.4 on the reference to parameters by name).
5. Each `inout`, `out` and `return` parameter of the activity maps to an output activity parameter node for the parameter. For each `inout` and `out` parameter, the activity includes an object flow from the assigned source, if any, for the parameter name after the activity block. For the further mapping for `return` parameters, see the mapping for return statements (in 18.13).

NOTE.

Even though activities are classes in UML, Alf does not provide any notation for defining attributes, operations or receptions for activities. However, in order to allow for the use of Alf representation for activities in larger enclosing models not represented in Alf, Alf allows activity model units to be active and to have attributes, operations and receptions as features. But, since, these features cannot be represented in Alf, it is tool specific how these features are attached to the activity.

Alf also does not provide any notation for specifying superclasses for an activity. However, Alf allows a tool to provide means for specifying the superclasses for an activity (which may be regular classes or activities) otherwise represented as an Alf model unit. Members are inherited from activity superclasses in the same way as for regular classes (sees 10.4.2 and 10.4.3). But the semantics for inheritance of behavior from activity superclasses is not specified.

19.12 Typed Element Definitions

1. A typed element definition maps to an element that is both a typed element and a multiplicity element, as given for the specific kind of typed element definition in the appropriate subsequent subclause.

Typed Element

2. The type of the typed element definition maps to the type of the typed element.

Multiplicity Element

3. The lower attribute of the multiplicity element is a literal integer for the value given by the lower attribute of the typed element definition.
4. The upper attribute of the multiplicity element is a literal unlimited natural for the value given by the upper attribute of the typed element definition.
5. The isUnique and isOrdered attributes of the multiplicity element are set according to the isNonUnique (with opposite sense) and isOrdered attributes of the typed element definition

19.13 Formal Parameters

1. A formal parameter maps to a parameter of an activity or an operation with the given name and direction. Its type and multiplicity are mapped as given in 19.12.

19.14 Property Definitions

1. A property definition maps to a property with the given name that is a structural feature of the classifier mapped from the classifier definition that is the namespace of the property definition. Its type and multiplicity are mapped as given in 19.12.
2. If the property definition is composite, then the property has aggregation=composite. Otherwise it has aggregation = none.
3. An initializer expression is not mapped as part of the property definition, but, rather, as part of the mapping of the constructor(s) for the owning class (see 19.5).

19.15 Operation Definitions

1. An operation definition maps to an operation with the given name and isAbstract value that is an owned operation of the class mapped from the class definition that is the namespace of the operation definition.
2. A formal parameter that is a member of an operation definition maps to an owned parameter of the operation (see 19.13).
3. If an operation declaration has redefined operations, then the operation has the these redefined operations.
4. If an operation definition has a body, then the operation has an associated activity (owned by the class of the operation) as its method, with the body mapped as if it was the body of an activity definition for this activity (see 18.1). The activity has an owned parameter corresponding, in order, to each owned parameter of the operation (see the mapping of formal parameters, below), with the same name, type, multiplicity and direction as the operation parameter.
5. If an operation definition is a stub, then its associated subunit maps to an activity. This activity becomes the method of the operation mapped from the operation definition.

Constructors

6. If the operation definition is a constructor, then it has an implicit return parameter with the owning class as its type. Further, the default constructor behavior (as described in 19.5) is included in the mapping of the operation body, sequentially before the explicit behavior defined for the constructor.

NOTE. If the method of a constructor operation is represented as an Alf statement sequence, but the operation is not itself represented using Alf textual notation, then the Alf standard does not specify the mapping of any behavior for the operation than that given in the explicit statement sequence. However, a modeling tool may insert additional behavior at the start of the constructor method, such as the default constructor behavior described above as part of the mapping of an Alf-represented constructor.

19.16 Reception Definitions

1. A reception definition maps to an reception with the given name and signal that is an owned reception of the active class mapped from the active class definition that is the namespace of the reception definition.

Annex A: Semantic Integration with State Machines and Composite Structure

(informative)

A.1 Overview

Subclause 6.2 describes the normative requirements for integrating the use of Alf text into the context of a larger UML model. In all such cases, the execution semantics of Alf are defined by the mapping of Alf to fUML. However, the UML models in which Alf may be used will likely often include constructs that are outside the fUML subset, with execution semantics as given in the UML Superstructure standard, but not covered by the fUML Specification. Some care must therefore be taken in understanding the semantic integration of Alf-specified behaviors with the constructs in which those behaviors are embedded.

This annex suggests how such an integration can be understood for the important cases of state machines (see UML Superstructure, Clause 15) and composite structure (see UML Superstructure, Clause 9). However, the approaches suggested here are not normative, both because the issues of this integration go beyond the basic requirements for the standard UML action language and because the UML Superstructure specification allows some variation in how this integration may be done. It is expected that normative semantic integration of state machines and composite structure with Alf (and fUML) will be addressed as a part of future standards that further formalize the execution semantics of those constructs.

A.2 State Machines

State machines are a commonly used mechanism for modeling state-dependent behavior in UML. Clause 15 of the UML Superstructure defines execution semantics for state machines. However, to be fully executable, it is necessary to also provide executable specifications for transition guard expressions and effect behaviors and state entry, exit and do-activity behaviors. One of the principal uses for UML action languages has been to specify such behaviors.

In this regard, it is to be expected that a common use of Alf will be to write the executable specifications of behaviors owned by state machines. Therefore, even though state machines are not themselves part of the fUML subset, it is important to understand how the formal execution semantics of Alf can be integrated with the semantics of state machines as given in Clause 15 of the UML Superstructure. An approach to such integration is described below for behaviors attached to state machine transitions and states.

B.2 provides an example of the use of Alf within a state machine model.

Transitions

Subclause 15.3.14 of the UML Superstructure defines the default notation for labeling a transition in a state machine. In the EBNF format used for Alf (see Table 6.1 in 6.4), this notation is:

```
Transition =
      [ Trigger { "," Trigger } [ "[" GuardConstraint "]" ]
      [ "/" BehaviorExpression ] ]
```

Subclause 13.3 of the UML Superstructure defines the textual notation for triggers of the various kinds of events. For the purposes of integration with Alf, guard constraints and behavior expressions may be defined as follows:

```
GuardConstraint = Expression
BehaviorExpression = Name | StatementSequence
```

with `Expression` as given in 8.1, `Name` as given in 7.6 and `StatementSequence` as given in 9.1.

The guard constraint and behavior expression in a transition are semantically mapped to fUML as follows, with the given constraints.

- A guard constraint expression must have type `Boolean`. The Alf expression for a guard constraint is mapped to an activity with a return parameter of type `Boolean` and no other parameters (as discussed in 8.1).
- If a behavior expression is given by a name, then this must be the name of the effect behavior owned by the transition. The named behavior may be specified by any means supported by the execution tool, in Alf or otherwise.
- If a behavior expression is given as an Alf statement sequence, then it maps to an activity (see 9.1).

Consistent with UML Superstructure, 15.3.14, the execution of a transition proceeds as follows:

1. If all the source states of the transition are in the active state configuration (or the source of the transition is a choice point that has been reached), one of the triggers of the transition is satisfied by the event occurrence being dispatched, and the transition has a guard constraint, then the guard constraint behavior is invoked synchronously. The execution object for the call is destroyed once the invocation is completed (see fUML Specification, 8.4.2.1, for an overview of execution objects in fUML). If the transition has a guard constraint that returns false, then the execution of the transition completes without further effect. Otherwise the transition is enabled to fire.
2. If the transition is enabled, then it may be selected to fire (per the rules of UML Superstructure, 15.3.14). If it is not selected, then its execution completes without further effect. Otherwise, execution proceeds as described below. (The description below only involves the execution of the transition behavior itself. For the specification of the effect of the transition on the state machine state configuration, see UML Superstructure, 15.3.14.)
3. If the transition has an effect behavior, then this behavior is invoked synchronously. The execution object for the invocation is destroyed once the invocation is completed (see fUML Specification, 8.4.2.1, for an overview of execution objects in fUML).

States

A state may optionally have an entry behavior, an exit behavior and/or a do-activity behavior. If a state has one or more of these behaviors, then this is notated in the internal activities compartment of the graphical symbol for the state as follows (see UML Superstructure, 15.3.11).

```
InternalActivity = ActivityLabel "/" BehaviorExpression
ActivityLabel = "entry" | "exit" | "do"
```

where `BehaviorExpression` is as given above.

The behavior expression in an internal activity specification is semantically mapped to fUML as follows, with the given constraints.

- If a behavior expression is given by a name, then this must be the name of the corresponding internal activity behavior owned by the state. The named behavior may have no parameters, and it may be specified by any means supported by the execution tool, in Alf or otherwise.
- If a behavior expression is given as an Alf statement sequence, then it maps to an activity with no parameters (see 9.1).

Consistent with UML Superstructure, 15.3.11, the execution of the internal activities of a state proceeds as follows.

1. If a state has an entry behavior, then this behavior is executed whenever the state is entered, before any other action is taken. The behavior is invoked synchronously and the execution object for the invocation is destroyed once the invocation is complete (see fUML Specification, 8.4.2.1, for an overview of execution objects in fUML).
2. If a state has a do-activity behavior, then this behavior begins executing whenever the state is entered, after the completion of the execution of the entry behavior (if any). The behavior is invoked asynchronously and it continues to execute while the state machine is in the state that owns it. When the owning state is exited for any reason, the do-activity execution object is destroyed (see fUML Specification, 8.4.2.1, for an overview of execution objects in fUML), which terminates the execution, if it is still on-going. If the do-activity behavior execution for a state terminates while the state machine is still in the owning state, and that state has an outgoing completion transition, then the state is exited, causing the do-activity execution object to be destroyed.
3. If a state has an exit behavior, then this behavior is executed whenever the state is exited, after the destruction of the do-activity execution object (if any). The behavior is invoked synchronously and the execution object for the invocation is destroyed once the invocation is complete (see fUML Specification, 8.4.2.1, for an overview of execution objects in fUML).

Current Event Data

An event occurrence that is dispatched for processing is known as the *current event* (see UML Superstructure, 13.3.31). If this is a signal or call event, then there may be data associated with the event occurrence: signal attribute values for a signal event or operation input parameter values for a call event. It is often necessary that transition and state behaviors to access such current event data for the event occurrences that trigger them.

There is currently no universally accepted, standard approach in UML for accessing current event data. This means that usually some tool-specific convention is adopted.

One popular approach is to make some sort of “current event variable” available within action language text, through which the current event data can be accessed. This works particularly well for signal events, for which such a variable can hold the signal instance for the current event. It is less well suited for call events, for which there is no specific single value associated with the call, and some convention is required for individually accessing operation parameters.

Unfortunately, there is no direct support in the UML action metamodel (let alone the fUML subset) for referring to any sort of current event variable. One related alternative approach is to provide access to current event data through calls to library functions, which can be invoked using normal call behavior actions. However, since such library functions cannot depend on specific signals and operations, the data they return had to be untyped, requiring the caller to do careful type testing and casting on the returned data.

Now, the UML Superstructure specification does actually include a textual notation for specifying the names to be associated with signal or call events (see the Notation sections of subclauses 13.3.6, 13.3.25, and 15.3.14). This notation is known as an *assignment specification:*

```
CallEvent = Name [ "(" AssignmentSpecification ")" ]
SignalEvent = Name [ "(" AssignmentSpecification ")" ]
AssignmentSpecification = AttrSpec [ "," AttrSpec ]
AttrSpec = Name [ ":" TypeName ]
```

An assignment specification may be included (but is not required) for a call event for an operation with parameters (other than a `return` parameter) or a signal event for a signal with attributes. The stated semantics of an assignment specification require that the given names be names of attributes of the context classifier for the state machine owning the transition and that the corresponding current event data are assigned to attributes so named.

However, the abstract syntax for events does not provide any way to capture such assignment specifications. Since UML semantics is formally defined on the abstract syntax, not any particular concrete syntax, there is no clear way to formalize the semantics given for the assignment specification notation.

One possible alternative is to use the assignment specification notation, but to treat the notation as specifying parameters with the given names on the *effect behavior* for the transition. Then, when that behavior is invoked, the current event data can be assigned to the behavior input parameters. For call events, the behavior would have parameters corresponding to both the input and output parameters of the called operation, with output parameters assigned after the behavior completes execution.

For this alternative, the effect behaviors on transitions would have to meet the following constraints.

- If a transition has more than one trigger or it has a single trigger that is not for a call event or a signal event, then its effect behavior (if any) may not have any parameters. If a transition has a single trigger for a call event or a signal event, then it may, but is not required, to have parameters, as given below.
- The effect behavior for a transition with a single trigger for a call event may have parameters that correspond, in order, to each of the parameters of the called operation. The direction, type and multiplicity of the behavior parameters must be the same as those of the corresponding operation parameters, but the names may be different.
- The effect behavior for a transition with a single trigger for a signal event may have `in` parameters that correspond, in order, to each of the attributes of the received signal. The type and multiplicity of the behavior parameters must be the same as those of the corresponding signal attributes, but the names may be different.

The assignment specification notation could then be used to show the names (and optionally the types) of the parameters of the effect behavior of a transition (other than the `return` parameter for a call event).

The advantage of this approach for Alf semantic integration is that, if an effect behavior with parameters as given above is specified using Alf, then the named parameters can be accessed as usual as local names within the Alf text (sees 8.3.3 and 8.8), and a return value (as appropriate for a call event) can be given using a `return` statement (see 9.14). A disadvantage is that these parameters are only available in the single effect behavior corresponding to the call event or signal event trigger. Input parameter values could be explicitly copied to context classifier attributes in order to make them to be available to, e.g., behaviors on downstream segments of a compound transition or on entry or exit behaviors. And there

would be no easy way at all to give values for output or return parameters outside of the original effect behavior.

A.3 Composite Structure

Clause 9 of the UML Superstructure defines the ability for modeling the composite structure of certain kinds of classifiers. This capability is widely used for modeling the hierarchical structure of large systems and systems of systems. It is also provides the basis for the structuring and encapsulation mechanisms commonly used with components (see UML Superstructure, Clause 8).

While neither composite structure nor components are part of the fUML subset, it is to be expected that, in larger models, executable behaviors will often be nested in some way within a component or other structured classifier. Fortunately, with certain restrictions, a composite structure model can be understood as an instruction for constructing a specific set of run-time objects connected by links, which may then be interpreted according to fUML semantics. This integration is described below.

B.3 provides an example of the use of Alf to specify executable behavior within the context of a composite structure model.

Parts and Connectors

A *part* of a structured classifier is simply a property that is an owned attribute of that classifier with composite aggregation. When an instance of a structured classifier is created, instances are created to fill the parts of the classifier, consistent with the multiplicity of those parts. Such instantiation may be specified using an instance model or through an explicit constructor operation. (See UML Superstructure, 9.3.13.)

The parts of a structured classifier may also act as the end points for connectors. A connector is a "specification of a link that enables communication between two or more instances" (UML Superstructure, 9.3.6). When an instance of a structured classifier is created, links are created between instances of parts of the classifier corresponding to any connectors between the parts, consistent with the multiplicity of the connector ends and of the parts (see UML Superstructure, 9.3.13).

A connector may optionally be typed by an association, in which case the links it specifies are instances of that association. According to UML Superstructure, 9.3.6, if the type of a connector that is omitted, the type is *inferred* based on the elements at the ends of the connector. The links specified by the connector are then instances of this inferred association, which does not actually appear in the model.

However, fUML semantics for links require that every link have an explicit association as its type unless it has been destroyed (see fUML Specification, 8.3.2.2.11). Therefore, for the purposes of integration with fUML execution, connectors must always have a modeled association as an explicit type. Further, as required in the fUML subset, every such association must own all its association ends.

NOTE. A tool does not actually have to require that a user explicitly create such a type for every connector. In fact, connector type does not even have to be an explicitly modeled association, as long as the tool does not compile Alf text to activity models and it provides some convention for naming the implicit connector type association (perhaps using the name of the connector or its ends). However, if Alf text navigating across a connector is compiled to an fUML-conformant activity, then the type of the connector must be physically in the model, so it can be referenced from appropriate actions in the activity, though it could be automatically created by the tool rather than having to be explicitly added by the user.

With the above restriction, once an instance of a structured classifier is constructed, its composite structure at run-time is simply the set of instances assigned to its parts, possibly interconnected by links of explicitly defined associations. While the fUML semantics do not provide a formal definition of how the construction of such an instance happens relative to the composite structure of its classifier, the resulting run-time structure is fully within the subset covered by fUML semantics. Therefore, Alf may be use to specify behavior related to the structured classifier based solely on modeling capabilities within the fUML subset, such as classes, attributes and associations.

For example, if a class `C` has a part `a` whose type is class `A` and a part `b` whose type is class `B`, with a connector between then whose type is association `R`, then the methods of operations of `C` can access parts a and b as a regular attributes of `C`. Further, behavior associated with class `A` can navigate as usual across association `R` in order to access the opposite instance of class `B`. And, for any instance that fills the part `C::a`, the opposite end of `R` is guaranteed by the semantics of composite structures and connectors to the instance that fills `C::b`—even though this is not formalized in fUML semantics, presumably this will be the semantics provided by any execution tool that supports UML composite structure.

Ports

A port represents an interaction point between an encapsulated classifier and its environment (see UML Superstructure, 9.3.11). The allowed interactions are specified using interfaces—*provided* interfaces specify requests the environment can make on instances of the classifier, while *required* interfaces specify request that instances of the classifier can make on the environment.

Ports are connectable elements. A connector that connects compatible ports on two internal parts of a structured classifier is known as an *assembly* connector. A connector that connects a port on a classifier to a compatible port on an internal part of that classifier is known as a *delegation connector*. (See UML Superstructure, 8.3.3, for the definition of assembly versus delegation connectors. Note that the connector kind is a derived attribute, based on the usage of the connector, not a flag that has to be set by the modeler.)

In addition, a delegation connector may be used to connect a port on a classifier directly to an internal part of that classifier. Requests to the provided interfaces of the port are then delegated directly to the internal part. Further, the internal part may send requests out through the required interfaces of the port.

Integration with fUML execution may be achieved by using fUML-subset capabilities to specify the behavior of an internal part connected to a port by a delegation connector. For the purposes of this integration, the part must realize all of the provided interfaces of the port, which must only have features that conform to the fUML subset. This allows requests received on the port to be delegated as normal invocations on the behavior of the internal part.

If a port does not have any required interfaces, then the delegation connector for it does *not* need to be explicitly typed by an association, since internal parts then have no need to send requests through the connector. However, if the port has one or more required interfaces, then the delegation connector must be explicitly typed by an association, where the type of the association end corresponding to the connector end connected to the port is a class that realizes all the *required* interfaces of the port. This allows requests sent out through the port to be modeled in the behavior of the internal part as normal invocations across the association.

NOTE. The specific requirements above should be considered to supersede the informally specified constraints in UML Superstructure, 9.3.6, on the compatibility of connectable elements at the ends of a connector with each other and with the types of the ends of the association typing the connector.

Interfaces and interface realization are not actually part of the fUML subset. However, a class that realizes an interface must itself own features that are compatible with all the features specified as part of the interface. Therefore, once the constraints of the interface realization are enforced, the realizing class implements within itself the features required by the interface and the interface realization relationship can then be ignored for the purposes of execution.

At run-time, an instance filling the internal part will be connected by a link (that is an instance of the association typing the delegation connector) to an instance of a class that realizes the required interfaces of the ports. The internal part instance may make calls on the opposite object over the link, and vice versa, per regular fUML semantics. Note that what instance is actually provided on the opposite end of a delegation link is execution tool specific—for example, it may be the ultimate end object or it may be an intermediate proxy object—but it must always have a type that realizes all the required interfaces of the delegated port.

For example, suppose a class has a port with a provided interface `P` that has an operation `x` and a required interface `R` that has an operation y and that this port is connected to an internal part whose type is class `A`. Then class `A` must realize interface `P` and have an operation that conforms to `P::x`. Further, the behavior of class `A` may navigate across the association typing the delegation connector in order to call the operation `R::y`.

Annex B: Extended Examples

(informative)

B.1 Quicksort Activity

This simple example presents the Quicksort algorithm in two forms: a "functional" form (see B.1.1) similar to how the algorithm might be naturally expressed in a functional programming language and an "in place" form (see B.1.2) that makes efficient use of memory. The intent is to show the flexibility of the Alf notation in expressing different algorithmic styles while maintaining a consistent underlying semantic mapping.

For completeness, both forms of the example are notated as complete Alf activity definitions. While such unit definitions are only available at the extended conformance level, the statement sequences bodies of these definitions could alternatively be attached to UML model as behavioral snippets, for example by including the Alf text as the body of an opaque behavior (see 9.1). In this case, the functional implementation would conform entirely to the full conformance level and the "in place" implementation would conform entirely to the minimum conformance level.

B.1.1 Quicksort Functional Implementation

The Quicksort activity noted below highlights the flow-oriented capabilities and concise OCL-like notation available in Alf at the full conformance level. Figure B.1 shows a graphical representation of this same activity.

```
activity Quicksort(in list: Integer[0..*] sequence):
  Integer[0..*] sequence                                    // See Note 1
{
  x = list[1];                                              // See Notes 2 & 3
  if (x == null) {                                          // See Note 4
    return null;                                            // See Notes 5 & 6
  } else {
    list1 = list->excludeAt(1);                             // See Note 7
    return Quicksort(list1->select a (a < x))->             // See Note 8
           including(x)->
           union(Quicksort(list1->select b (b >= x)));
  }
}
```

Notes

1. Alf allows the use of the usual UML multiplicity notation (see 10.5.2). The keyword "`sequence`" indicates that the input list is an ordered sequence of integers that allows repetition of values.
2. Alf uses the C-like notation of "=" for assignment (and "==" for equality). However, Alf does not require that the types for local names be explicitly declared (see 9.6). The type for `x` is determined implicitly here as `Integer`, the result type of the expression being assigned to the name. Alternatively, Alf also allows explicit type declaration, either in the form "`let x: Integer = list[1];`" or the C-like form "`Integer x = list[1];`".
3. Alf provides a familiar index notation for accessing the elements of a sequence (see 8.3.16). Note, however, that indexing is from 1, not 0. Also, if the index value is either less than 1 or greater than the current size of the sequence, the result is not an error, but, instead, the sequence index expression

evaluates to `null`. Thus, in the statement "`x = list[1];`", x will be assigned the value `null` if `list` is empty.

4. An Alf `if` statement has a C-like syntax and similar semantics (see 9.8). In addition, because of the test for whether the local name `x` is `null`, it is known to be non-null in the `else` clause of the `if` statement.
5. As would be expected, a return statement is used in Alf to provide the return value for an activity (see 9.14).
6. In Alf, "`null`" denotes the empty set (see 8.3.15).
7. An OCL-like notation can be used for *sequence operations* (see 8.3.17) such as `excludeAt`.
8. Alf provides on OCL-like notation for *sequence expansion expressions* (see 8.3.19). The expression "`list->select a (a < x)`" is used to select all elements of list that are less than x. Note that there is no required ordering of the tests on the elements of list, which may all be carried out concurrently.

   There are two major notational differences from OCL. First, the notation for an iterator variable is different. Instead of the OCL form "`select(a | ...)`", Alf uses "`select a (...)`". This avoids ambiguity with the use of "`|`" for the Boolean "or" operator in Java/C syntax. Second, Alf requires that an iterator variable always be given. Java-like expression syntax is not as rigorously object-oriented as OCL syntax, and, therefore, not as amenable to an implicit form for the iterator context. By requiring that the iterator variable be explicit, no special form is necessary for the expressions used within the collection operation construct. For example, the OCL expression "`devices->select(oclIsKindOf(Sensor))`" is written in the form "`devices->select device (device instanceof Sensor)`" in Alf.

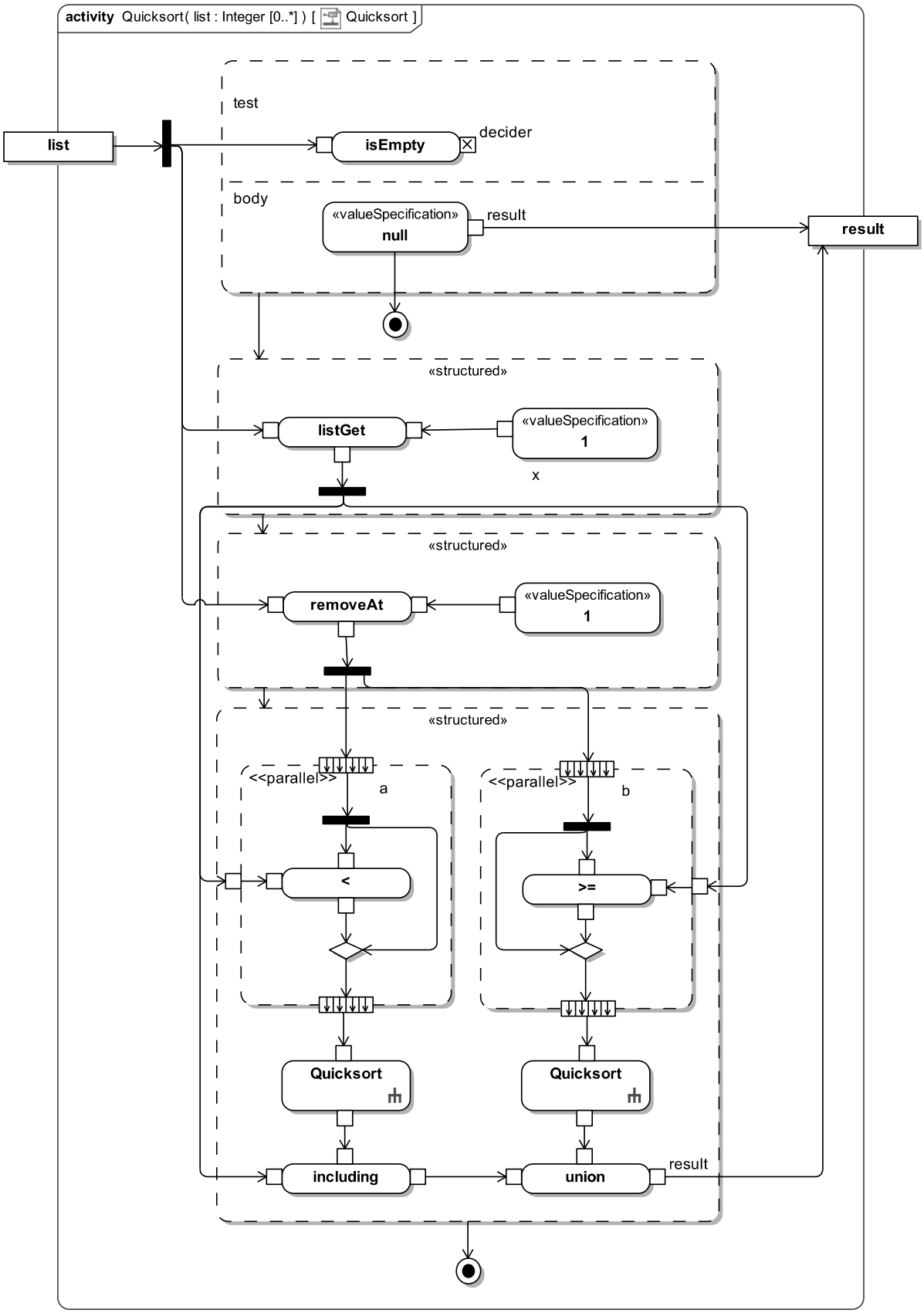

Figure B.1 Activity Diagram for `Quicksort`

B.1.2 Quicksort "In Place" Implementation

The `QuicksortInPlace` activity notated below highlights the flexibility of Alf to all both a functional or procedural implementation of the algorithm equally naturally. The procedural notation used in this form is entirely available at the minimum conformance level and is very similar syntactically to code in C or Java. (The equivalent graphical representation of this activity is too complicated to easily show here.)

```
activity QuicksortInPlace
  (inout list: Integer[0..*] sequence,                    // See Note 1
   in low: Integer, in high: Integer)
{
  if (low < high) {
    l = low;                                              // See Note 2
    h = high;
    p = list[high];

    do {                                                  // See Note 3
      while ((l < h) && (list[l] <= p == true)) {         // See Notes 4 & 5
        l = l+1;
      }
      while ((h > l) && (list[h] >= p == true)) {
        h = h-1;
      }
      if (l < h) {
        t = list[l];
        list[l] = list[h];
        list[h] = t;
      }
    } while (l < h);

    t = list[l];
    list[l] = list[high];
    list[high] = t;

    QuicksortInPlace(list, low, l-1);
    QuicksortInPlace(list, l+1, high);
  }
}
```

Notes

1. Alf allows activity parameters with any of the UML directions, including `out` and `inout` (see 10.4.8).
2. Statements within the statement sequence of a block delimited by "`{…}`" are executed sequentially (see 9.1).
3. An Alf `do…while` statement has the familiar C-like syntax and semantics (see 9.11).
4. Alf uses the usual infix notation for arithmetic and relational operations (sees 8.6.1 and 8.6.4). The *conditional and* operator "`&&`" has the C-like semantics of only evaluating its second argument if the first argument is true (see 8.6.8).
5. If the index value in a *sequence index expression* such as "`list[l]`" is less than zero or greater than the size of the indexed sequence, the expression evaluates to `null` (see 8.3.16). This means that, in the relational expression "`list[l] <= p`", both `list[l]` and p may evaluate to `null` and, therefore, have multiplicity lower bound of 0, and, as a result, so does the relational expression. However, the

operands of the conditional-and operator `&&` are required to both have multiplicity lower bound of 1 (i.e., `null` values are not allowed), which is why "`list[1] <= p`" is tested for equality to `true` in the second operand of the conditional-and expression. If the relational expression happens to evaluate to `null`, then this is *not* equal to `true`, so the `null` result is effectively treated as `false`.

B.2 Online Bookstore

B.2.1 Graphical Model for Ordering

This example is adapted from (a portion of) the Online Bookstore Domain Case Study given in Appendix B of the book *Executable UML: A Foundation for Model Driven Architecture* by Stephen J. Mellor and Marc J. Balcer (Addison-Wesley, 2002).

This example will focus only on the `Ordering` package for the Online Bookstore. Figure B.2 shows a class diagram for this package.

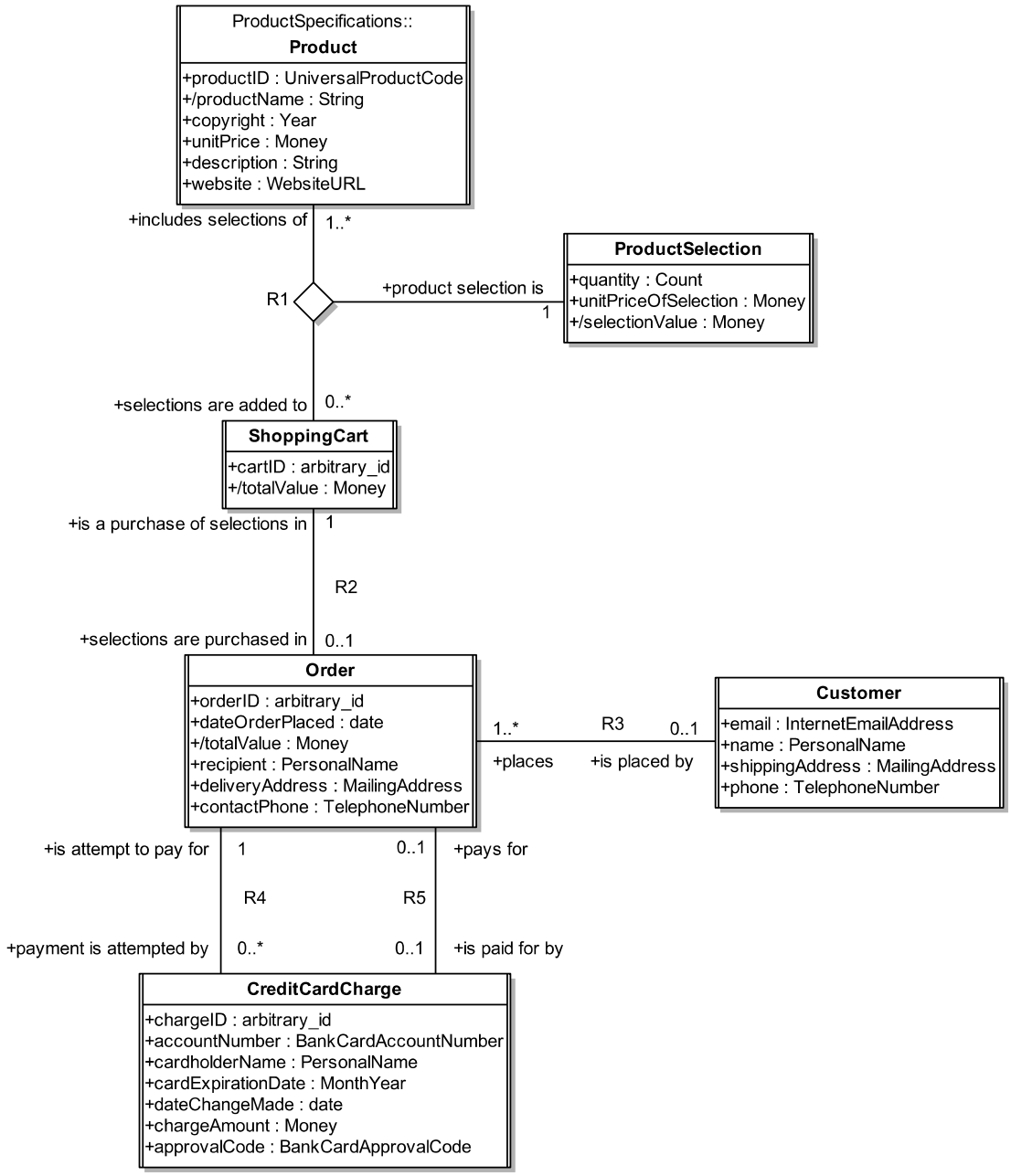

Figure B.2 Ordering Subsystem Class Diagram

As indicated in Figure B.2, each of the classes shown is an active class. This means that each one has a classifier behavior triggered by a number of signal events. Only the class `Order` from the `Ordering` package will be further detailed in this example. Figure B.3 shows the state machine that is the classifier behavior for class `Order`.

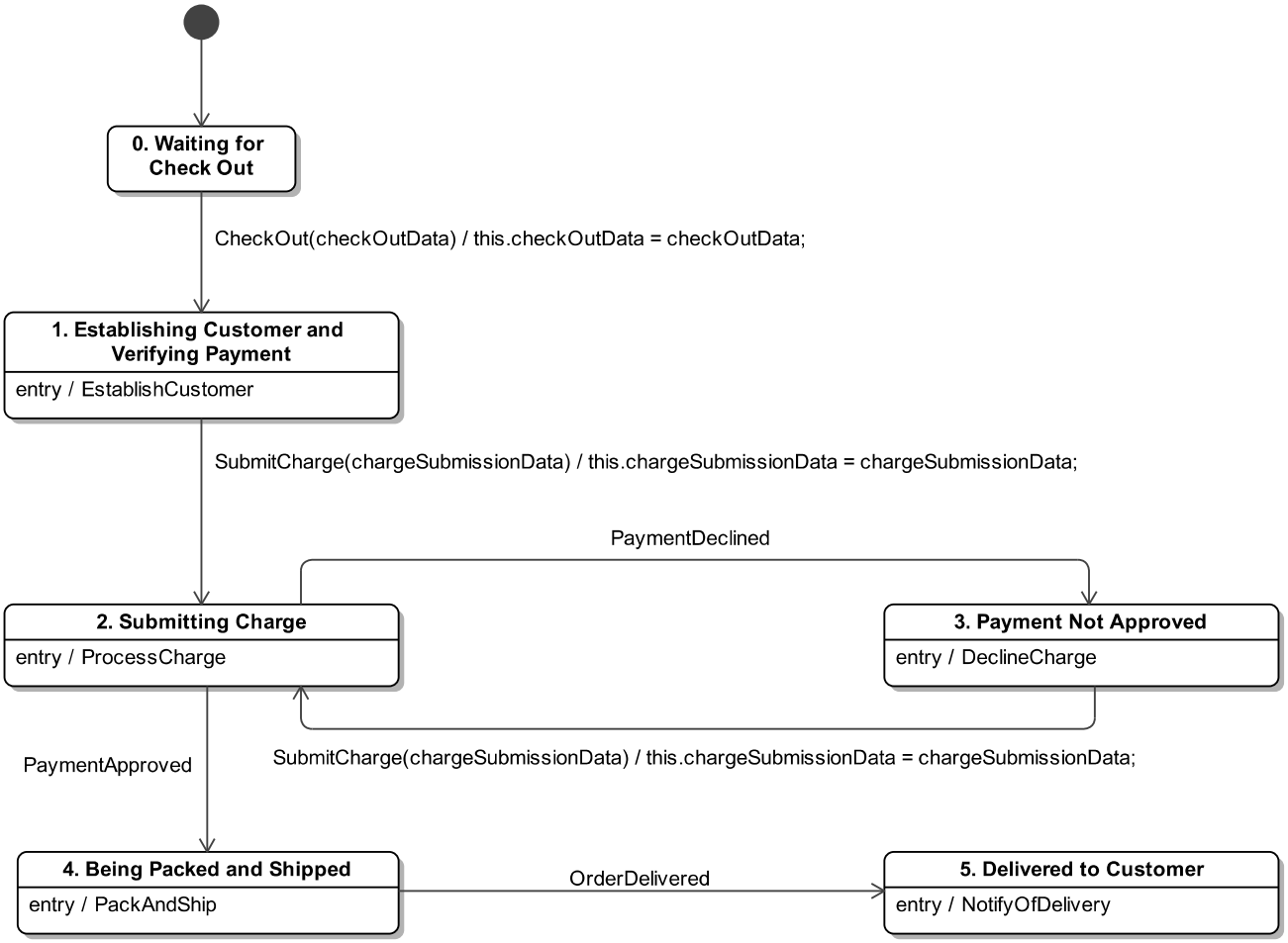

Figure B.3 State Machine Classifier Behavior for Class Order

The triggers on the transitions in this state machine are all for signal events. Of the five different signals shown, only `CheckOut` and `SubmitCharge` carry data. For simplicity, each of these signals has a single attribute whose type is a data type that contains the appropriate data for the signal.

As discussed in A.2, the trigger notation "`CheckOut(checkOutData)`" is used here to indicate that the effect behavior for the transition has a single in parameter corresponding to the single attribute of the `CheckOut` signal. As shown in Figure B.3 the effect behavior for the transition assigns the parameter to a correspondingly named attribute of class `Order`, which can then be accessed by the entry behavior of the target state. A similar strategy is used for the transitions triggered by the `SubmitCharge` signal. (Note that a tool could provide support for automatically generating effect behaviors with such attribute assignments and then suppressing the behavior from the diagram, thus effectively providing the "assignment specification" semantics intended in the UML Superstructure, 15.3.14.)

Figure B.4 shows a complete model for class `Order`, including the (private) attributes `checkOutData` and `chargeSubmissionData`. This model also includes receptions (indicated by the «`signal`» keyword)

that declare the ability of `Order` to receive each of the signals used in the triggers in the state machine shown in Figure B.3. Note that these receptions have the same names as their respective signals.

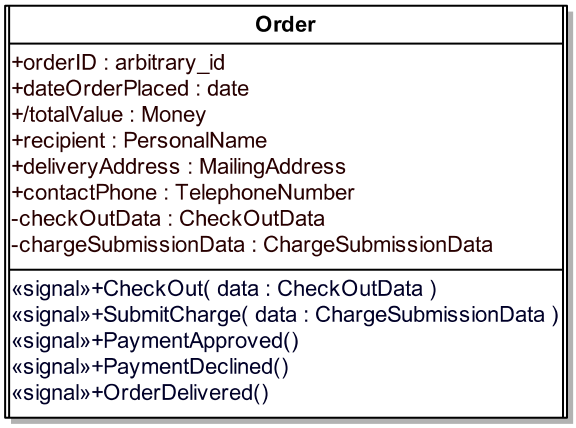

Figure B.4 Active Class `Order`

Each of the states shown in Figure B.3 has an entry behavior that is executed whenever the state is entered. The bodies of these entry behaviors are specified below using Alf notation. The names shown in the entry notation are the names of the behaviors (see also A.2 on this notation).

B.2.2 Alf Representation of Entry Behaviors

This subclause includes an Alf representation for each of the entry behaviors identified in Figure B.3. Each behavior is defined as an Alf unit for an activity that is attached to the appropriate state as its entry behavior.

NOTE. As discussed in 9.1, a modeling tool could also define each of the entry behaviors as opaque behaviors, with the Alf text for the behavior as the body of the opaque behavior. In this case, the Alf text would only include the statement sequences from the activity definitions given below, not the full unit definition, which would not require the extended conformance level notation for units.

B.2.2.1 Activity EstablishCustomer

```
private import TIM;
/**                                                        // See Note 1
Entry action for State 1. Establishing Customer and Verifying Payment
*/
activity EstablishCustomer()
{
  R2.createLink (                                          // See Note 2
    'selections are purchased in'    => this,
    'is a purchase of selections in' => this.checkOutData.cart
  );

  // Create a Customer if one does not already exist
  // with the given email address
  matchingCustomers = Customer -> select c                 // See Note 3
                        (c.email == this.checkOutData.customerEmail);

  if (matchingCustomers->isEmpty()) {
    customer = new Customer();                             // See Note 4
    customer.email = this.checkOutData.customerEmail;
  } else {
    customer = matchingCustomers[1];                       // See Note 5
  }

  // Use the name, address, etc. to update Customer
  // whether new or existing
  customer.name = this.checkOutData.customerName;
  customer.shippingAddress = this.checkOutData.shippingAddress;
  customer.phone = this.checkOutData.customerPhone;

  // Link the order to the customer
  R3.createLink (
    places          => this,
    'is placed by'  => customer
  );

  // Set the date order placed to today
  this.dateOrderPlaced = TIM::GetCurrentDate();

  // Create a credit card charge and submit it
  // to the credit card company
  this.SubmitCharge(                                       // See Note 6
    new ChargeSubmissionData (                             // See Note 7
      accountNumber      => this.checkOutData.accountNumber,
      billingAddress     => this.checkOutData.billingAddress,
      cardExpirationDate => this.checkOutData.cardExpirationDate,
      cardholderName     => this.checkOutData.cardholderName
    )
  );
}
```

Notes

1. Text (possibly across multiple lines) bracketed by "`/**`" and "`*/`" denotes a documentation comment that is attached to the model element in which it appears. On the other hand, text that begins with "`//`" is a lexical comment that is not mapped into any model element.

2. Links are instances of associations. The notation "`R1.createLink`" indicates the creation of a new link of the association `R1`, with the following tuple giving the association end data. This maps to a create link action.
3. The class name "`Customer`" here denotes the current extent of that class. In this context, it is a shorthand for the notation "`Customer.allInstances()`". The `select` expression then selects the customers with e-mail addresses matching the one given in the received event.
4. There is no constructor operation defined for class `Customer`, so the notation "`new Customer()`" is a "constructorless" instance creation expression the simply creates a new instance of `Customer`. Further, since `Customer` is an active class, the active behavior of the new instance is also automatically started.
5. Alf uses an array-like notation for indexing ordered collections. In this case, the notation "`matchingCustomers[1]`" denotes getting the first element of the list of matching customers. Note that lists are indexed starting at 1.
6. Alf uses the same invocation notation for sending a signal as for calling an operation. Sending a signal is indicated by the selected feature ("`SubmitCharge`" in this case) being the name of a reception of the given target object ("`this`").
7. The `SubmitCharge` signal has a single attribute of type `ChargeSubmissionData`. Since `ChargeSubmissionData` is a data type, an instance expression for it has as its arguments values for each of the attributes of the newly created data value. Alf allows for a named-parameter notation, used here, in which the names "`accountNumber`", "`billingAddress`", etc. are interpreted as the names of arguments for constructing the `ChargeSubmissionData` value.

B.2.2.2 Activity ProcessCharge

```
private import Ordering::CreditCardCharge::ChargeData;
/**
Entry behavior for State 2. Submitting Charge
*/
activity ProcessCharge()
{
  // Create a Credit Card Charge and submit it
  // to the credit card company
  creditCardCharge = new CreditCardCharge();
  creditCardCharge.MakeCharge(
    new ChargeData (
      accountNumber      => this.chargeSubmissionData.accountNumber,
      billingAddress     => this.chargeSubmissionData.billingAddress,
      cardExpirationDate => this.chargeSubmissionData.cardExpirationDate,
      chargeAmount       => this.totalValue,
      order              => this
    )
  );
}
```

B.2.2.3 Activity DeclineCharge

```
private import EE_OnlineCustomer;
/**
Entry behavior for State 3. Payment Not Approved
*/
activity DeclineCharge()
{
  // Notify the customer that the charge was rejected
  customer = this.'is placed by';
  EE_OnlineCustomer.ChargeDeclined(customerEmail => customer.email);
}
```

B.2.2.4 Activity PackAndShip

```
private import EE_OnlineCustomer;
private import Shipping::Shipment;
/**
Entry behavior for State 4. Being Packed and Shipped
*/
activity PackAndShip()
{
  // Notify the customer that the charge was approved
  // and the order will be shipped
  customer = this.'is placed by';
  EE_OnlineCustomer.ChargeApproved(customerEmail => customer.email);

  // Create a shipment to send the order to the customer
  shipment = new Shipment();
  shipment.RequestShipment(order => this);
}
```

B.2.2.5 Activity NotifyOfDelivery

```
private import EE_OnlineCustomer;
/**
Entry behavior for State 5. Delivered to Customer
*/
activity NotifyOfDelivery()
{
  // Notify the customer that the Order
  // has been delivered
  customer = this.'is placed by';
  EE_OnlineCustomer.OrderReportedDelivered(customerEmail => customer.email);
}
```

B.2.3 Alf Representation of the Ordering Model

The previous subclauses describe a typical scenario in which an overall UML model is represented largely graphically, with various behavioral snippets represented textually in Alf. However, at the extended conformance level, Alf also includes notation for structural models, such as packages, classes and associations (see Clause 10). This subclause shows how the models for the package Ordering and the class Order, represented graphically in B.2.1, can alternatively be represented textually in Alf.

B.2.3.1 Package Ordering

```
private import EE_OnlineCustomer;
private import ProductSpecification::Product;
private import DomainDataTypes::*;                          // See Note 1

/**
Online Bookstore, Ordering Subsystem
*/
package Ordering
{
  public active class ShoppingCart;                         // See Note 2
  public active class Order;
  public active class CreditCardCharge;
  public active class Customer;
  public active class ProductSelection;

  public assoc R1
  {
    public 'selections are added to': ShoppingCart[0..*];  // See Note 3
    public 'includes selections of':  Product[1..*];
    public 'product selection is':    ProductSelection;
  }

  public assoc R2 {
    public 'selections are purchased in':    Order[0..1];
    public 'is a purchase of selections in': ShoppingCart;
  }

  public assoc R3 {
    public places:         Order[1..*];
    public 'is placed by': Customer[0..1];
  }

  public assoc R4 {
    public 'is an attempt to pay for': Order;
    public 'payment is attempted by':  CreditCardCharge[0..*];
  }

  public assoc R5 {
    public 'pays for':      Order[0..1];
    public 'is paid for by': CreditCardCharge[0..1];
  }
}
```

Notes

1. The clause "`private import ProductSpecification::Product;`" denotes an element import of the `Product` class in the `ProductSpecification` package. Use of "`private`" indicates a private import—"`public import ProduceSpecification::Product;`" would indicate a public import. Similarly. "`private import DomainDataTypes::*;`" denotes the (private) package import of the package `DomainDataTypes`. That is, it is not just the package `DomainDataTypes` that is being imported, but all the elements of that package.

2. All the classes in this package have classifier behaviors and are thus active. Only stubs are included in this package specification, with the full definition of the class written separately. The use of stubs

is a general option to allow long namespace definitions to be split up into multiple files. In contrast to the class definitions, the shorter association definitions are included in-line in this example.

3. Names do not have to conform to the usual identifier lexical structure. Such general names are enclosed in single quotes, as in "`'selections are added to'`". Note that every character within the quotes is significant in the name (including spaces, but tabs and line breaks are not allowed).

B.2.4 Class Order

```
namespace Ordering;

/**
Active class for managing an order
*/
active class Order
{
  public orderID:               arbitrary_id;
  public dateOrderPlaced:       date;
  public totalValue:            Money;
  public recipient:             PersonalName;
  public deliveryAddress:       MailingAddress;
  public contactPhone:          TelephoneNumber;

  private checkOutData:         CheckOutData;
  private chargeSubmissionData: ChargeSubmissionData;

  public datatype CheckOutData
  {
    public cart:                ShoppingCart;
    public accountNumber:       BankCardAccountNumber;
    public billingAddress:      MailingAddress;
    public cardExpirationDate:  MonthYear;
    public cardholderName:      PersonalName;
    public customerEmail:       InternetEmailAddress;
    public customerName:        PersonalName;
    public customerPhone:       TelephoneNumber;
    public shippingAddress:     MailingAddress;
  }

  public datatype ChargeSubmissionData
  {
    public accountNumber:       BankCardAccountNumber;
    public billingAddress:      MailingAddress;
    public cardExpirationDate:  MonthYear;
    public cardholderName:      PersonalName;
  }

  public receive signal CheckOut                            // See Note 1
  {
    public data:                CheckOutData;
  }

  public receive signal SubmitCharge
  {
    public data:                ChargeSubmissionData;
  }
```

```
  public receive signal PaymentDeclined{}
  public receive signal PaymentApproved{}
  public receive signal OrderDelivered{}

  private EstablishCustomer();                              // See Note 2
  private ProcessCharge();
  private DeclineCharge();
  private PackAndShip();
  private NotifyOfDelivery();

}
do                                                          // See Note 3
{

  /** 0. Waiting for Check Out */                           // See Note 4
  accept (checkOut: CheckOut);                              // See Note 5
  this.checkOutData = checkOut.data;

  /** 1. Establishing Customer and Verifying Payment */
  this.EstablishCustomer();                                 // See Note 6

  while (true) {                                            // See Note 7

    accept (chargeSubmission: SubmitCharge);
    this.chargeSubmissionData = chargeSubmission.data;

    /** 2. Submitting Charge */
    this.ProcessCharge();

    accept (PaymentDeclined) {
      /** 3. Payment Not Approved */
      this.DeclineCharge();

    } or accept (PaymentApproved) {                         // See Note 8
      break;                                                // See Note 9
    }

  }

  /** 4. Being Packed and Shipped */
  this.PackAndShip();
  accept (OrderDelivered);

  /** 5. Delivered to Customer */
  this.NotifyOfDelivery();

}
```

Notes

1. The notation "`receive signal Checkout`" indicates the definition of both a signal called `Checkout` within the `Order` namespace *and* the definition of a reception of that signal as a feature of the `Order` class. A signal definition alone would be denoted "`signal Checkout{…}`"and a definition of a reception for that signal would be "`receive Checkout`". Note that, unlike the graphical UML notation, in Alf the attributes ("parameters") of the signal would not be repeated in the reception definition.

2. Rather than standalone activities, the effective "entry behaviors" are now specified as privately owned operations of class `Order`. However, other than the different containing namespace, the definitions of the activities that provide the methods for these operations are exactly the same as given in B.2.2. (Note that fUML semantics ensures that the correct context object for "`this`" references is propagated to operations called from the body of a classifier behavior.)
3. The "`do`" part of an active class definition gives its classifier behavior. Since Alf does not provide a notation for state machines, the classifier behavior for `Order` is specified here as an equivalent activity.
4. A documentation comment in this position attaches to the model element mapped from the following statement.
5. The "`accept`" statement waits for reception of the indicated signal and then continues. The name `checkOut` is given to the signal instance that is received.
6. Alf uses the usual object-oriented procedural notation for operation invocation.
7. The `while` statement provides structured iteration.
8. This is a *compound* accept statement. The additional "`or accept`" clause indicates waiting for reception of *either* a `PaymentDeclined` *or* a `PaymentApproved` signal. If a `PaymentDeclined` signal is received, then the statements associated with that accept clause ("`this.DeclineCharge();`") are executed. If a `PaymentAccepted` signal is received, then the statements associated with its accept clause ("`break;`") are executed.
9. The `break` statement terminates the enclosing `while` loop.

B.3 Property Management Service

B.3.1 The Property Management Model

The following is an example of using Alf to specify the methods for operations within the context of a larger model. This context is a simplified model for a service-oriented architecture for property management (where the term "property" is used here in the sense of "tangible asset"—e.g., furniture, cars, buildings, etc.). Two points in particular should be kept in mind while reading this example.

First, a guiding principle for Alf is that use of Alf should place little or no restriction on the larger model in which Alf is used. For this example, it should be supposed that the graphical model was created before it was decided to use Alf to specify activities for operation methods, so Alf had no influence on the stylistic choices for the model. For example, the following modeling capabilities used in the example model go beyond what is often used in software-oriented class models, but are not precluded by the use of Alf:

- Names with spaces
- Data types
- Associations with owned ends

Second, Alf should be usable in the context of models not limited to the fUML subset. Indeed, this example uses UML modeling constructs outside that context. Of course, any portion of the model actually specified in Alf is effectively limited to the fUML subset, but this should not prevent integration with the larger model. For example, the following UML capabilities used in the example model are not in the fUML subset, but are compatible with the use of Alf for the specification of activities (though they could not actually be represented using Alf's textual notation for structural models).

- Default values
- Derived values
- Constructors
- Interfaces
- Components
- Ports, Parts and Connectors

(See also the discussion in A.3 on the use of Alf in the context of composite structure models.)

This subclause describes the `Property Management` model that provides the context for the Alf specification of activities given in B.3.5. This model has the overall package structure shown in Figure B.5.

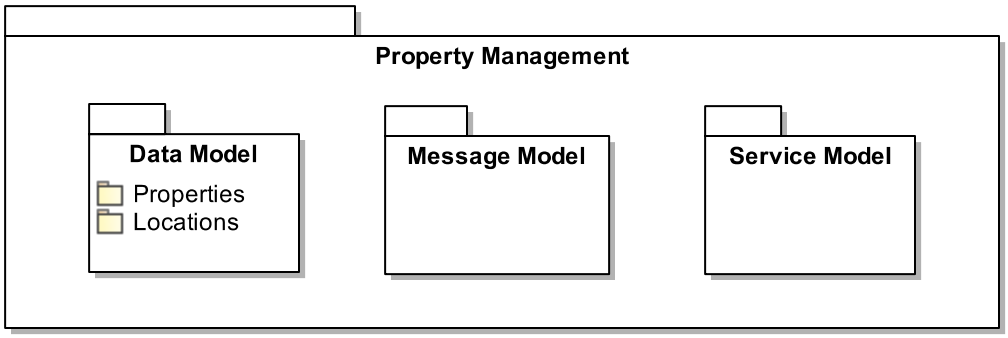

Figure B.5 `Property Management` Package Diagram

Each of the three packages under Property Management is discussed in turn in the following subclauses.

B.3.2 Data Model

A data model defines the persistent *entities* in a domain and the data necessary to describe them. The data maintained on an entity serves as a *record* of that entity for service being provided. Clearly, `Property` is the fundamental entity for which records are being kept by the `Property Management Service`.

Figure B.6 gives a simplified data model for property management. Note in general the use of spaces in names. This is allowed in a UML model and is not uncommon in business-oriented data models not primarily driven by an information technology style.

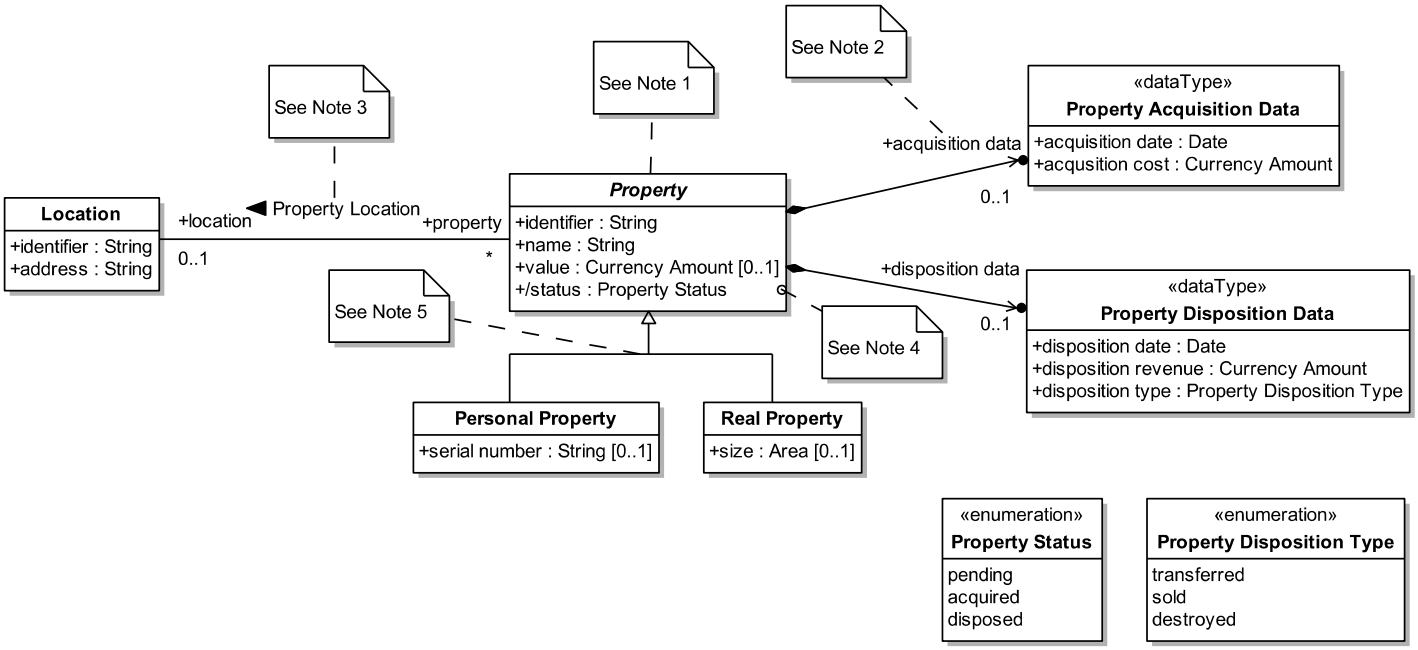

Figure B.6 Property Data Model

Notes

1. The `Property` class models the record of a specific property. A class is used because a property record has the semantics of an object, with an identity and attributes that may change over time.

However, the property record is also given a unique domain `identifier` that is used to externally identify the record.

2. `Property Acquisition Data` and `Property Disposition Data` are considered to be simply composite attributive data of a property record, without independent identity. They are therefore modeled as data types. The association ends `acquisition data` and `disposition data` are owned by `Property` and are therefore attributes of `Property`.
3. Unlike `Property Acquisition Data` and `Property Disposition Data`, `Location` is a class. It is supposed here that there is a problem domain concept of an identifiable location and that there may be multiple properties at the same location (note the "`*`" multiplicity on the `property` association end). The association `Property Location` owns both its association ends.
4. The `Property::status` attribute is derived. The derivation constraint is given in OCL as:

```
context Property inv property_status_derivation:
  (acquisition_data -> isEmpty() implies status = pending) and
  (acquisition_data -> notEmpty() and disposition_data -> isEmpty()
    implies status = acquired) and
  (disposition_data -> notEmpty() implies status = disposed)
```

   (Note that underscores are used here to fill in the spaces in the names. The OCL Specification does not seem to disallow spaces in names, but it is not clear how they would parse in the OCL syntax.)
5. The `Property` class is abstract with two concrete subclasses for `Personal Property` (property that is movable, like vehicles and furniture) and `Real Property` (property that is immovable, like land and buildings). The attributes of these subclasses are optional, because their value is not necessarily known until the property is actually acquired (a property record can be established pending acquisition, in order to generate a property identifier with which to track the acquisition).

In addition to the data structure shown in Figure B.6, the `Property` class has two operations. As shown in Figure B.7, one operation is a constructor and the other handles the computation of the derived value for the `status` attribute.

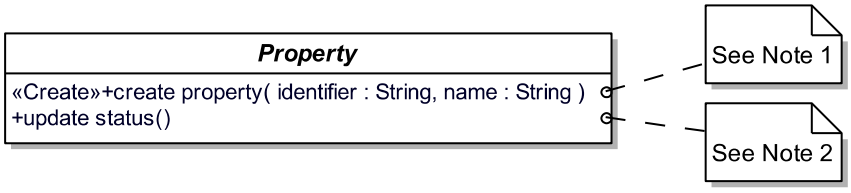

Figure B.7 Property Operations

Notes

1. The stereotype `«create»` is the standard UML notation used to annotate a constructor operation. Note that there is no particular naming convention required (e.g., the name of the constructor does *not* have to be the same as the name of the class). The `Property::identifier` and `name` attributes are required, so values for these are given as arguments to the constructor (the setting of the `status` attribute, which is also required, is discussed below). However, the constructor operation does not have any return type shown (per the conventions shown in 9.3.1 of the UML Superstructure), though it has an implicit return type of `Property`.

2. The `update status` operation is used to update the derived `status` attribute consistent with its derivation constraint (given earlier). This operation is called by the constructor, but it is also used by the service operations given later.

B.3.3 Message Model

In a *service oriented architecture* (SOA), a request message is sent to initiate a transaction to carry out the requested action. Once the transaction for a request has completed, a reply is sent back to the requester indicating the success or failure of the transaction.

B.3.3.1 Request Messages

The Property Management Service being discussed here allow the following requests:

- `Property Record Establishment`: Create a property record pending acquisition of the property.
- `Property Acquisition Notification`: Add or update the acquisition data for an existing property record. The property status must not be `disposed`.
- `Property Record Update`: Update the attributes of an existing property record not related to acquisition or disposal. The property status must not be `disposed`.
- `Property Disposition Notification`: Add or update the disposition data for an existing property record. The property status must be `acquired`.
- `Property Record Deletion`: Delete a property record.
- `Property Record Retrieval`: Retrieve a property record given the identifier for the property.

Figure B.8 shows a model of the messages for these requests.

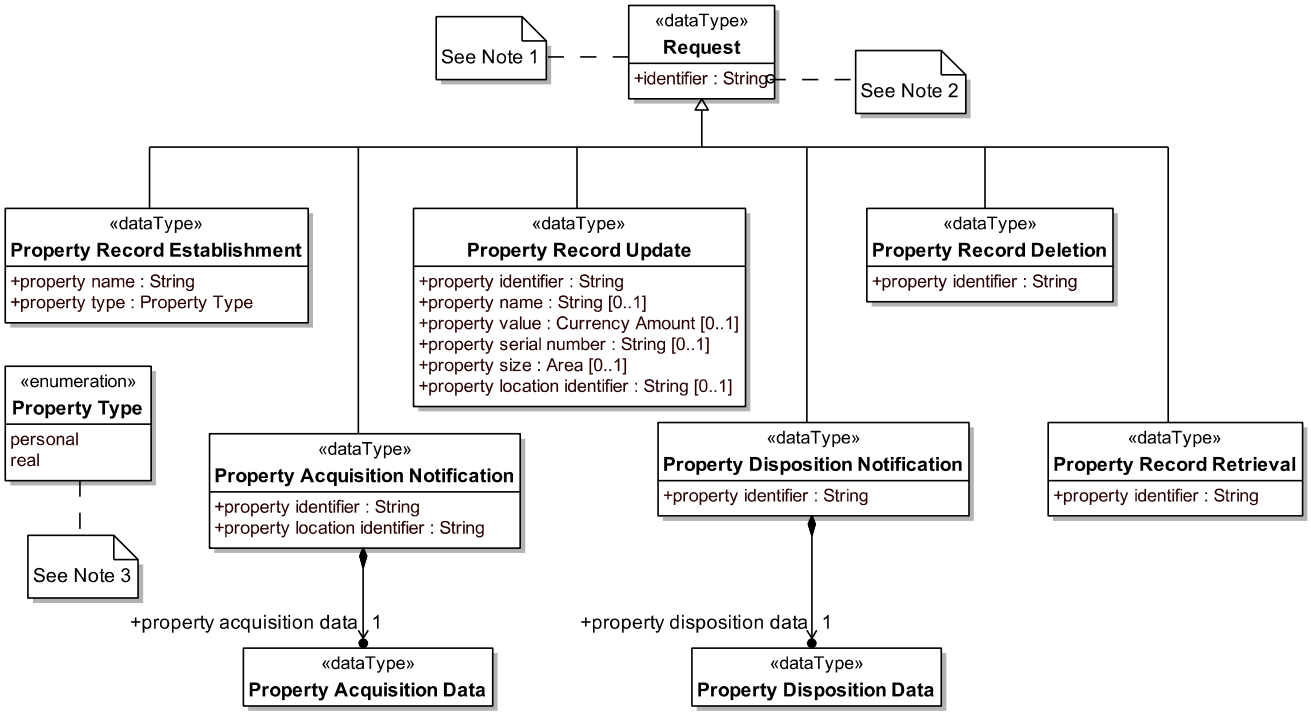

Figure B.8 Request Message Model

Notes

1. All messages are modeled as data types. Messages are always passed by value, not by reference.
2. Despite being data types, every message has a sender-assigned `identifier`. For a request message, the identifier is used to correlate reply messages (see below).
3. For the purposes of this example, the different types of properties are identified in the messages using enumerated values, rather than by specialization.

B.3.3.2 Reply Messages

Replies are also delivered using messages, as modeled below.

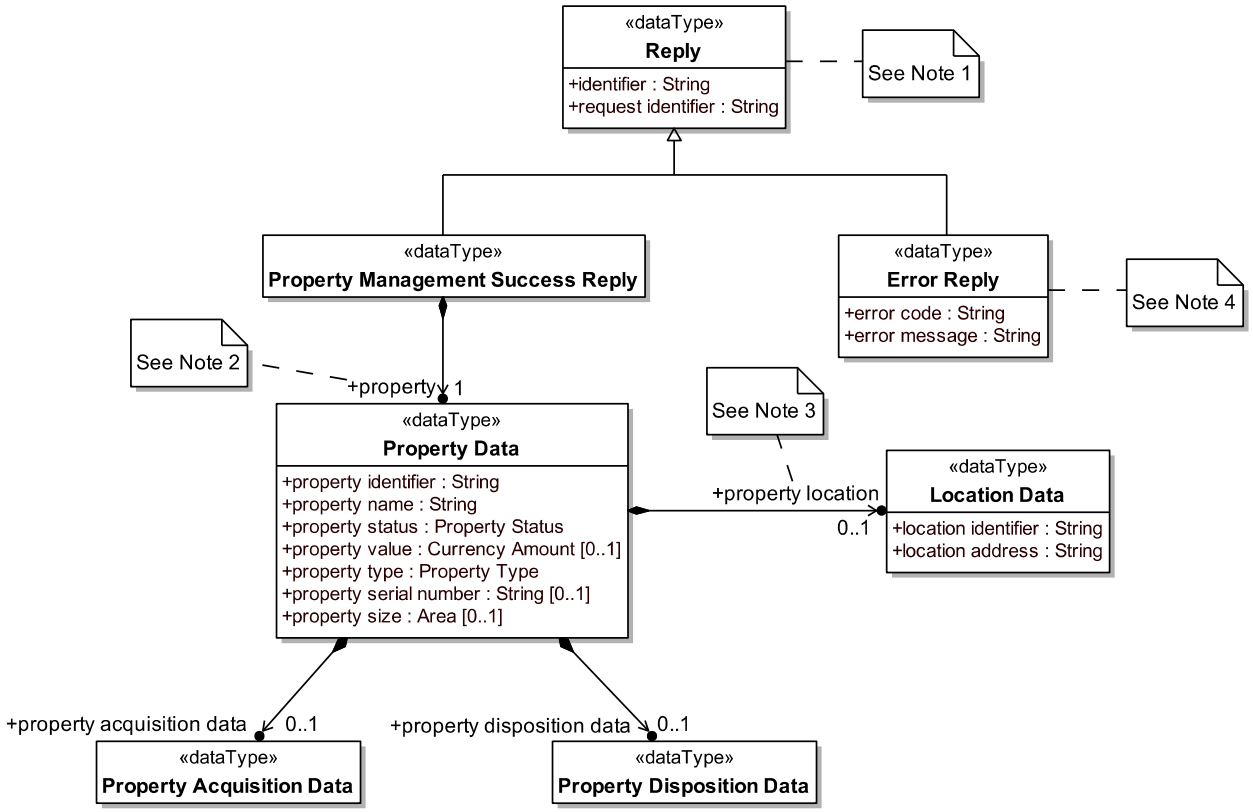

Figure B.9 Reply Message Model

Notes

1. As with request messages, reply messages are modeled as data types with `identifiers`. Every reply message also has a `request identifier` message that identifies the request message to which the reply correlates.
2. For the purposes of this example, all requests, if successful, reply by returning a copy of the identified property record. In the case of a record removal, this is a copy of the removed record. (A more realistic model would likely have different reply messages corresponding to different requests.)
3. Messages cannot contain object references. Therefore, all relevant data from the location object is composed into the reply message.

4. For the purposes of this example, all requests, if failed, reply by returning a generic `Error Reply`.

B.3.4 Service Model

The interface for the `Property Management Service` can now be defined as an interface with operations to deliver each of the request messages defined above. This interface is provided by a `Property Management Service Provider` component that uses a `Property Management Service Implementation` class to implement the operations on the service interface.

Figure B.10 shows the `Property Management Service` interface. For the purposes of this example, each operation has a simple synchronous request/reply signature. Each operation takes a single in parameter `request` of a specific request type and two `out` parameters. If the request is successful, a success reply is returned in the `reply` parameter. Otherwise an error reply is returned in the `error` parameter.

Figure B.10 `Property Management Service` Interface

Figure B.11 shows the composite structure of the `Property Management Service Provider` component.

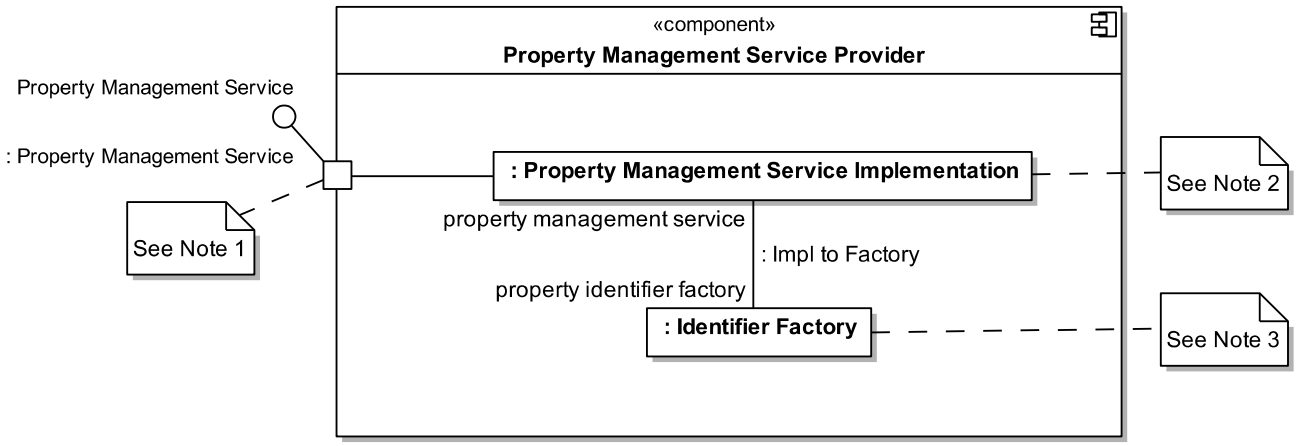

Figure B.11 `Property Management Service Provider` Component Composite Structure

Notes

1. The `Property Management Service` is used to type a port on the `Property Management Service Provider`, becoming the single provided interface for the port.
2. The `Property Management Service Provider` has an internal part that directly implements the service. All requests through the port are delegated to this implementation.

3. The `Property Management Service Provider` has another part that is used by the `Property Management Service Implementation` as a "factory" for creating unique identifiers to give to property records when they are established.

Figure B.12 shows a model of the Property Management Service Implementation class.

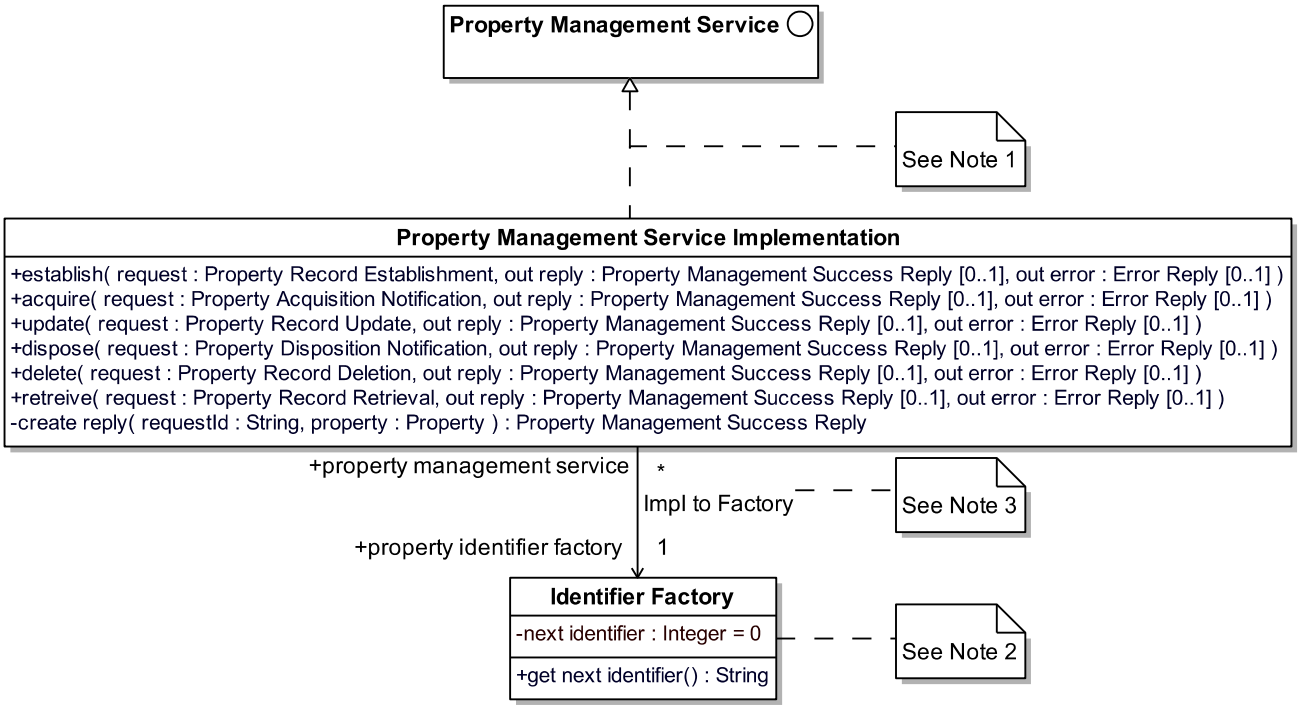

Figure B.12 `Property Management Service Implementation` Class

Notes

1. The `Property Management Service Implementation` class realizes the `Property Management Service` interface, implementing all its operations. In addition, it adds a private operation used to create reply messages and an association to the generic `Identifier Factory` class (which is used to connect it to the `Identifier Factory` part within the `Property Management Service Provider` component, as shown earlier).
2. The `Identifier Factory` has the simple behavior of generating unique identifiers as sequential integers starting from 0.
3. The `Impl to Factory` association acts as the type of the connector shown in Figure B.11.

B.3.5 Property Management Service Methods

This subclause provides Alf specifications of activities that serve as the methods of all the operations shown on classes in the previous section.

B.3.5.1 Property

As discussed in B.3.2, the class `Property` has two operations, `create property`, which is a constructor, and `update status`, which computes the derived value of the `status` attribute.

create property

```
namespace 'Property Management'::'Data Model'::Properties::Property;
                                                          // See Note 1

/** Create a new property with a given identifier and name */
activity 'create property'(in identifier: String, in name: String) {

  this.identifier = identifier;
  this.name = name;
  this.'update status'();

}
```

Notes

1. The namespace clause here includes the fully qualified name for the `Property` class, uniquely identifying it within the model. Note also the use of single quotes to form names with spaces.

update status

```
namespace 'Property Management'::'Data Model'::Properties::Property;

/** Update the status of a property consistent with whether it has been
    acquired or disposed.
*/
activity 'update status'() {
  if (this.'acquisition data' -> isEmpty()) {              // See Note 1
    this.status = 'Property Status'::pending;              // See Note 2
  } else if (this.'disposition data' -> isEmpty()) {
    this.status = 'Property Status'::acquired;
  } else {
    this.status = 'Property Status'::disposed;
  }
}
```

Notes

1. The OCL-like notation "`this.'acquisition data' -> isEmpty()`" is used to test whether the optional attribute `acquisition data` is empty.
2. The enumeration `Property Status` is visible in the scope of `Property` (it is in the same package), so it can be used without import. However, the names of its enumeration literals still need to be qualified.

B.3.5.2 Identifier Factory

The `Identifier Factory` class has a single operation used to get the next unique identifier.

get next identifier

```
namespace Identifiers::'Identifier Factory';

/** Generate a unique identifier from the next sequential integer. */
activity 'get next identifier'(): String {
  return IntegerFunctions::ToString(this.'next identifier'++);  // See Note 1
}
```

Notes

1. Alf allows the use of the C-style "++" operator. In this case, the value of next identifier is incremented, but the *prior* value is returned. That is, the return statement above is mapped as if it were the following equivalent statement sequence:

```
this.'next identifier' = (oldValue = this.'next identifier') + 1;
return IntegerFunctions::toString(oldValue);
```

B.3.5.3 Property Management Service Implementation

The operations in the class `Property Management Service Implementation` handle each of the service requests modeled in B.3.3. In addition, the class has one private operation, `create reply`, which is a utility operation used to construct a reply message.

Note in particular that fUML semantics requires that objects persist between activity invocations at a specific execution locus, unless they are explicitly destroyed. The specification of the `Property Management Service` takes advantage of this by effectively using the extent of the `Property` class as a "database" of property records.

create reply

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
                                                          // See Note 1
private import 'Property Management'::'Message Model'::*;

/** Create a reply message for a given message ID and property object */
activity 'create reply'(in requestId: String, in property: Property):
                                  'Property Management Success Reply' {

  propertyData = new 'Property Data' (                    // See Note 2
    'property identifier'       => property.identifier,
    'property name'             => property.name,
    'property status'           => property.status,
    'property acquisition data' => property.'acquisition data',
    'property disposition data' => property.'disposition data'
    'property type'             => property instanceof 'Personal Property'?
                                     'Property Type'::personal:
                                     'Property Type'::real );

  if (property.location -> notEmpty()) {
    propertyData.'property location'
      = new 'Location Data' (
          'location identifier' => property.location.identifier,
          'location address'    => property.location.address );
  }

  if (property instanceof 'Personal Property') {
    propertyData.'property serial number'
      = (('Personal Property')property).'serial number';
  } else {
    propertyData.'property size' = (('Real Property')property).size;
  }
```

```
  return new 'Property Management Success Reply' (
    identifier           => requestId + "/reply",
    'request identifier' => requestId,
    property             => propertyData );
}
```

Notes

1. The notation "`private import 'Property Management'::'Data Model'::Properties::*`" indicates a package import. That is, all the elements of the package `'Property Management'::'Data Model'::Properties` are visible within the activity `create reply`.
2. `Property Data` is a data type, which may only be constructed using a default constructor. The default constructor for a data type provides arguments for each of the attributes of the data type, which are set here using a named-parameter notation.

establish

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Message Model'::*;

/** Establish a new property record. */
activity establish (
  in request: 'Property Record Establishment',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  identifier = this.'property identifier factory'.'get next identifier'();
                                                          // See Note 1

  if (request.'property type' == 'Property Type'::personal) {
    property = new 'Personal Property'::'create property'
                 (identifier, request.'property name');
                                                          // See Note 2
  } else {
    property = new 'Real Property'::'create property'
                 (identifier, request.'property name');
  }

  reply = this.'create reply'(request.identifier, property);
}
```

Notes

1. The notation "`this.'property identifier factory'`" maps to a read link action for the association from the `Property Management Service Implementation` to the `Identifier Factory` used to generate property identifiers.

   NOTE. Components and parts are not in the fUML subset. However, by general UML semantics, it is assumed that, when the `Property Management Service Provider` is instantiated, both its parts are also instantiated. The connection between the parts then results in a link between the `Property Management Service Implementation` instance and the `Instance Factory` instance. It is this link that is read here. (See also the discussion in A.3.)

2. The `Property` class has the named constructor `create property`, rather than a default constructor. This is referenced in the constructor invocation by the qualified name "`Property::'create property'`". Note that if, instead, the constructor had had the same name as the class, then the constructor could have been referenced simply using the class name, i.e., "`new Property(identifier, request.name)`".

acquire

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Message Model'::*;

/** Update the acquisition information for an existing property. */
activity acquire (
  in request: 'Property Acquisition Notification',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  property =
    Property -> select p (p.identifier == request.'property identifier')[1];
                                                          // See Note 1

  if (property -> isEmpty()) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PAN-001",
      'error message'      => "Property not found." );

} else {
    location = Location -> select loc
                 (loc.identifier == request.'property location identifier')[1];
    if (location -> isEmpty()) {
      error = new 'Error Reply' (
        identifier           => request.identifier + "/error",
        'request identifier' => request.identifier,
        'error code'         => "PAN-003",
        'error message'      => "Location not found." );
    } else {
      'Property Location'.createLink(property, location);
      property.'acquisition data' = request.'property acquisition data';
                                                          // See Note 2
      property.'update status'();

      reply = this.'create reply'(request.identifier, property);
    }
}
```

Notes

1. The `select` expression is used to select the elements from a collection that meet the given condition. The type name "`Property`" here is used as a shorthand for "`Property.allInstances()`"—that is, the extent of the `Property` class. Thus, this expression selects the instance of `Property` (if any) whose identifier equals that given in the request.

2. This is an assignment to the `acquisition data` attribute for `property`.

   NOTE. `acquisition data` is an opposite association end owned by the class `Property`. The fUML subset does not actually include such associations (fUML requires that all ends of associations are owned by the association). However, since an association end owned by a class is a structural feature, it can be written by an add structural feature value action. But, per fUML semantics, no link is actually created—only the structural feature is set.

update

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Data Model'::Locations::Location;
private import 'Property Management'::'Message Model'::*';

/** Update the attribute data of a property (other than acquisition or
    disposition data). Only non-empty values in the update message cause
    corresponding attribute updates. Note that none of these updates can
    result in a property status change.
*/
activity update (
  in request: 'Property Record Update',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  property = Property -> select p
               (p.identifier == request.'property identifier')[1];

  if (property -> isEmpty()) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PRU-001",
      'error message'      => "Property not found." );

  } else if (property.status == 'Property Status'::disposed) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PRU-002",
      'error message'      => "Property already disposed." );

  } else if ((request.'property serial number' -> notEmpty() &&
              property instanceof 'Real Property') ||
             (request.'property size' -> notEmpty() &&
              property instanceof 'Personal Property')) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PRU-002",
      'error message'      => "Wrong property type." );

    } else {
    if (request.'property location identifier' -> notEmpty()) {
      location = Location -> select loc
```

```
                  (loc.identifier == request.'property location identifier');
      if (request.'property location identifier' -> notEmpty()) {
        location = Location -> select loc
                     (loc.identifier == request.'property location identifier')[1];
        if (location -> isEmpty()) {
          error = new 'Error Reply' (
            identifier           => request.identifier + "/error",
            'request identifier' => request.identifier,
            'error code'         => "PRU-003",
            'error message'      => "Location not found." );
          return;
        } else {
          'Property Location'.createLink(property, location);
        }
      }
    }
    property.name = request.'property name' ?? property.name;
    property.value = request.'property value' ?? property.value;

    if (property instanceof 'Personal Property') {
      property.'serial number' =
            request.'property serial number' ?? property.'serial number';
    }

    if (property instanceof 'Real Property') {
      property.size = request.'property size' ?? property.size;
    }

    reply = this.'create reply'(request.identifier, property);
  }
}
```

dispose

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Message Model'::*;

/** Update the disposition data for an existing property. */
activity dispose (
  in request: 'Property Disposition Notification',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  property = Property -> select p
               (p.identifier == request.'property identifier')[1];

  if (property -> isEmpty()) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PDN-001",
      'error message'      => "Property not found." );

  } else if (property.status == 'Property Status'::pending) {
```

```
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PDN-002",
      'error message'      => "Property not yet acquired." );

  } else {
    property.'disposition data' = request.'property disposition data';
    property.'update status'();

    reply = this.'create reply'(request.identifier, property);
  }
}
```

delete

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Message Model'::*;

/** Delete an existing property, destroying the record of it. */
activity delete (
  in request: 'Property Record Deletion',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  property = Property -> select p
               (p.identifier == request.'property identifier')[1];

  if (property -> isEmpty()) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PRD-001",
      'error message'      => "Property not found." );

  } else {
    reply = this.'create reply'(request.identifier, property);
    property.destroy();                                    // See Note 1

  }
}
```

Note

1. The expression "`property.destroy()`" destroys the object `property`.

retrieve

```
namespace 'Property Management'::'Service Model'::
                           'Property Management Service Implementation';

private import 'Property Management'::'Data Model'::Properties::*;
private import 'Property Management'::'Message Model'::*;

/** Retrieve data on an existing property record. */
```

```
activity retrieve (
  in request: 'Property Record Retrieval',
  out reply: 'Property Management Success Reply' [0..1],
  out error: 'Error Reply' [0..1] ) {

  property = Property -> select p
               (p.identifier == request.'property identifier')[1];

  if (property -> isEmpty()) {
    error = new 'Error Reply' (
      identifier           => request.identifier + "/error",
      'request identifier' => request.identifier,
      'error code'         => "PRR-001",
      'error message'      => "Property not found." );

  } else {
    reply = this.'create reply'(request.identifier, property);

  }
}
```

B.4 Alf Standard Library Collection Classes Implementation

B.4.1 Introduction

This example provides a sample implementation for the Alf Standard Library `CollectionClasses` package in Alf itself. As discussed in 11.7, implementations for these classes are supplied in the nested `CollectionClasses::Impl` package. B.4.2 presents the Alf representation of the `CollectionClasses::Impl` package, with the following subclauses giving Alf units for each of the classes contained in this package.

Note that this is not intended to be an efficient, production implementation of the collection class package. Rather, it is intended to provide a concrete example of the use of the `Impl` mechanism for the collection classes and to demonstrate that all the behavior specified for the collection classes can be implemented in Alf using other features.

B.4.2 CollectionClasses::Impl

The `CollectionClasses::Impl` package is required to contain the classes shown in Figure 11.6. The Alf implementation given for the package below also includes two additional classes that are used to factor out common behavior from the collection classes (other than `Map`). The package is also marked as a model library, so that it does not automatically import the Alf standard library itself (see 10.1).

```
namespace Alf::Library::CollectionClasses;
package Impl {

  private abstract class CollectionImpl<T>;
  private abstract class OrderedCollectionImpl<T> specializes CollectionImpl<T>;

  public class Set<T>
    specializes CollectionImpl<T>, CollectionClasses::Set<T>;
  public class Bag<T>
    specializes CollectionImpl<T>, CollectionClasses::Bag<T>;

  public class OrderedSet<T>
```

```
    specializes OrderedCollectionImpl<T>, CollectionClasses::OrderedSet<T>;
  public class List<T>
    specializes OrderedCollectionImpl<T>, CollectionClasses::List<T>;

  public class Queue<T>
    specializes CollectionImpl<T>, CollectionClasses::Queue<T>;
  public class Deque<T>
    specializes Queue<T>, CollectionClasses::Deque<T>;

  public class Map<Key,Value> specializes CollectionClasses::Map<Key,Value>;

}
```

B.4.3 CollectionClasses::Impl::CollectionImpl

The abstract `CollectionImpl` class provides implementations for the operations (other than `toSequence`) common to all collection classes other than `Map`. The actual collection content is expected to be supplied by the concrete subclasses of `CollectionImpl`. The `CollectionImpl` operation method use abstract methods `setContent` and `toSequence`, to be defined in the subclasses, to set and get this content.

Note that, since the `Impl` package is stereotyped as a model library, the usual importation of standard library packages is suppressed. Therefore, the `CollectionFunctions` package must be explicitly imported in order to make the collection functions usable without qualification. The collection functions are imported privately, as required by the rules of 11.7 for the implementation of collection classes.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The base concrete implementation for all the standard library collection classes.
*/
abstract class CollectionImpl<T> {

  @Create protected CollectionImpl(in seq: T[0..*] sequence) {
    this.setContent(seq);
  }

  protected abstract setContent (in seq: T[0..*] sequence);

  public abstract toSequence (): T[0..*] sequence;

  public add (in element: T): Boolean {
    result = this.excludes(element);
    this.setContent(this.toSequence()->including(element));
    return result;
  }

  public addAll (in seq: T[0..*] sequence): Boolean {
    preSize = this.size();
    this.setContent(this.toSequence()->union(seq));
    return this.size() > preSize;
  }

  public clear () {
    this.setContent(null);
  }
```

```
public count (in element: T): Integer {
  return this.toSequence()->count(element);
}

public equals (in seq: T[0..*] sequence): Boolean {
  return this.toSequence()->equals(seq);
}

public excludes (in element: T): Boolean {
  return this.toSequence()->excludes(element);
}

public excludesAll (in seq: T[0..*] sequence): Boolean {
  return this.toSequence()->excludesAll(seq);
}

public includes (in element: T): Boolean {
  return this.toSequence()->includes(element);
}

public includesAll (in seq: T[0..*] sequence): Boolean {
  return this.toSequence()->includesAll(seq);
}

public isEmpty (): Boolean {
  return this.toSequence()->isEmpty();
}

public notEmpty (): Boolean {
  return this.toSequence()->notEmpty();
}

public remove (in element: T): Integer {
  result = this.count(element);
  this.setContent(this.toSequence()->excluding(element));
  return result;
}

public removeAll (in seq: T[0..*] sequence): Boolean {
  preSize = this.size();
  this.setContent(this.toSequence()->difference(seq));
  return this.size() < preSize;
}

public removeOne (in element: T): Boolean {
  result = this.includes(element);
  this.setContent(this.toSequence()->excludingOne(element));
  return result;
}

public replace (in element: T, in newElement: T): Integer {
  result = this.count(element);
  this.setContent(this.toSequence()->replacing(element, newElement));
  return result;
}
```

```
  public replaceOne (in element: T, in newElement: T): Boolean {
    result = this.includes(element);
    this.setContent(this.toSequence()->replacingOne(element, newElement));
    return result;
  }

  public retainAll (in seq: T[0..*] sequence): Boolean {
    preSize = this.size();
    this.setContent(this.toSequence()->intersection(seq));
    return this.size() < preSize;
  }

  public size (): Integer {
    return this.toSequence()->size();
  }

}
```

B.4.4 CollectionClasses::Impl::OrderedCollectionImpl

The class `OrderedCollectionImpl` extends `CollectionImpl` with implementation of additional operations found in the ordered collection classes (i.e., `OrderedSet` and `List`).

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The base concrete implementation for the standard library ordered collection
classes
*/
abstract class OrderedCollectionImpl<T> specializes CollectionImpl<T> {

  @Create protected OrderedCollectionImpl(in seq: T[0..*] sequence) {
    super(seq);
  }

  public addAllAt (in index: Integer, in seq: T[0..*] sequence): Boolean {
    preSize = this.size();
    this.setContent(this.toSequence()->includeAllAt(index, seq));
    return this.size() > preSize;
  }

  public addAt (in index: Integer, in element: T): Boolean {
    return this.addAllAt(index, element);
  }

  public at (in index: Integer): T[0..1] {
    return this.toSequence()->at(index);
  }

  public first (): T[0..1] {
    return this.at(1);
  }

  public indexOf (in element: T) : Integer[0..1] {
    return this.toSequence()->indexOf(element);
  }
```

```
  public last (): T[0..1] {
    return this.at(this.size());
  }

  public removeAt (in index: Integer): T[0..1] {
    result = this.at(index);
    this.setContent(this.toSequence()->excludeAt(index));
    return result;
  }

  public replaceAt (in index: Integer, in element: T): T[0..1] {
    result = this.at(index);
    this.setContent(this.toSequence()->replacingAt(index, element));
    return result;
  }
}
```

B.4.5 CollectionClasses::Impl::Set

The `Set` class given below implements the abstract `CollectionClasses::Set` class. It does this by defining a `content` attribute of the appropriate type with unbounded multiplicity. By default, the attribute is unique and unordered—that is, a set. Operation methods are provided by specializing the `CollectionImpl` class.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template Set class.
*/
class Set<T> specializes CollectionImpl<T>, CollectionClasses::Set<T> {

  private content: T[0..*];

  @Create public Set (in seq: T[0..*] sequence) {
    super.CollectionImpl(seq);
  }

  @Destroy public destroy () {  }

  private setContent (in seq: T[0..*] sequence) {
    this.content = seq;
  }

  public add (in element: T): Boolean {
    return super.CollectionImpl<T>::add(element);
  }

  public addAll (in seq: T[0..*] sequence): Boolean {
    return super.CollectionImpl<T>::addAll(seq->toOrderedSet());
  }

  public clear () {
    super.CollectionImpl<T>::clear();
  }

  public count (in element: T): Integer  {
    return super.CollectionImpl<T>::count(element);
```

```
}

public equals (in seq: T[0..*] sequence): Boolean {
  set = seq->toOrderedSet();
  return this.size() == set->size() && this.includesAll(set);
}

public excludes (in element: T): Boolean {
  return super.CollectionImpl<T>::excludes (element);
}

public excludesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::excludesAll(seq->toOrderedSet());
}

public includes (in element: T): Boolean {
  return super.CollectionImpl<T>::includes(element);
}

public includesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::includesAll(seq->toOrderedSet());
}

public isEmpty (): Boolean {
  return super.CollectionImpl<T>::isEmpty();
}

public notEmpty (): Boolean {
  return super.CollectionImpl<T>::notEmpty();
}

public remove (in element: T): Integer {
  return super.CollectionImpl<T>::remove(element);
}

public removeAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::removeAll(seq);
}

public removeOne (in element: T): Boolean {
  return super.CollectionImpl<T>::removeOne(element);
}

public replace (in element: T, in newElement: T): Integer  {
  return super.CollectionImpl<T>::replace(element, newElement);
}

public replaceOne (in element: T, in newElement: T): Boolean {
  return super.CollectionImpl<T>::replaceOne(element, newElement);
}

public retainAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::retainAll(seq);
}

public size (): Integer {
  return super.CollectionImpl<T>::size();
```

```
  }

  public toSequence (): T[0..*] sequence {
    return this.content;
  }

}
```

B.4.6 CollectionClasses::Impl::OrderedSet

The `OrderedSet` class given below implements the abstract `CollectionClasses::OrderedSet` class. It does this by defining a `content` attribute of the appropriate type with unbounded multiplicity. The attribute is specifically defined to be ordered and is unique by default—that is, it is an ordered set. Operation methods are provided by specializing the `OrderedCollectionImpl` class.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template OrderedSet class.
*/
class OrderedSet<T>
  specializes OrderedCollectionImpl<T>, CollectionClasses::OrderedSet<T> {

  private content: T[0..*] ordered;

  @Create public OrderedSet (in seq: T[0..*] sequence) {
    super.OrderedCollectionImpl(seq->toOrderedSet());
  }

  @Destroy public destroy () {
  }

  private setContent(in seq: T[0..*] sequence) {
    this.content = seq;
  }

  public add (in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::add(element);
  }

  public addAt (in index: Integer, in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::addAt(index, element);
  }

  public addAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::addAll(seq->toOrderedSet());
  }

  public addAllAt (in index: Integer, in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::addAllAt(index, seq->toOrderedSet());
  }

  public at (in index: Integer): T[0..1] {
    return super.OrderedCollectionImpl<T>::at(index);
  }

  public clear () {
```

```
    super.OrderedCollectionImpl<T>::clear();
  }

  public count (in element: T): Integer  {
    return super.OrderedCollectionImpl<T>::count(element);
  }

  public equals (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::equals(seq->toOrderedSet());
  }

  public first (): T[0..1] {
    return super.OrderedCollectionImpl<T>::first();
  }

  public indexOf (in element: T) : Integer[0..1] {
    return super.OrderedCollectionImpl<T>::indexOf(element);
  }

  public last (): T[0..1] {
    return super.OrderedCollectionImpl<T>::last();
  }

  public excludes (in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::excludes (element);
  }

  public excludesAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::excludesAll(seq->toOrderedSet());
  }

  public includes (in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::includes(element);
  }

  public includesAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::includesAll(seq->toOrderedSet());
  }

  public isEmpty (): Boolean {
    return super.OrderedCollectionImpl<T>::isEmpty();
  }

  public notEmpty (): Boolean {
    return super.OrderedCollectionImpl<T>::notEmpty();
  }

  public remove (in element: T): Integer {
    return super.OrderedCollectionImpl<T>::remove(element);
  }

  public removeAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::removeAll(seq);
  }

  public removeAt (in index: Integer): T[0..1] {
    return super.OrderedCollectionImpl<T>::removeAt(index);
```

```
  }

  public removeOne (in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::removeOne(element);
  }

  public replace (in element: T, in newElement: T): Integer {
    return super.OrderedCollectionImpl<T>::replace(element, newElement);
  }

  public replaceAt (in index: Integer, in element: T): T[0..1] {
    result = this.at(index);
    if (result->notEmpty()) {
      this.remove(result);
      this.addAt(index, element);
    }
    return result;
  }

  public replaceOne (in element: T, in newElement: T): Boolean {
    return super.OrderedCollectionImpl<T>::replaceOne(element, newElement);
  }

  public retainAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::retainAll(seq);
  }

  public size (): Integer {
    return super.OrderedCollectionImpl<T>::size();
  }

  public subOrderedSet (in lower: Integer, in upper: Integer):
CollectionClasses::OrderedSet<T> {
    return new OrderedSet<T>(this.toSequence()->subsequence(lower, upper));
  }

  public toSequence(): T[0..*] sequence {
    return this.content;
  }

}
```

B.4.7 CollectionClasses::Impl::Bag

The `Bag` class given below implements the abstract `CollectionClasses::Set` class. It does this by defining a `content` attribute of the appropriate type with unbounded multiplicity. The attribute is specifically defined to be nonunique and is ordered by default—that is, it is a bag. Operation methods are provided by specializing the `CollectionImpl` class.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template Bag class.
*/
class Bag<T> specializes CollectionImpl<T>, CollectionClasses::Bag<T> {

  private content: T[0..*] nonunique;
```

```
@Create public Bag (in seq: T[0..*] sequence) {
  super.CollectionImpl(seq);
}

@Destroy public destroy () { }

private setContent(in seq: T[0..*] sequence) {
  this.content = seq;
}

public add (in element: T): Boolean {
  return super.CollectionImpl<T>::add(element);
}

public addAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::addAll(seq);
}

public clear () {
  super.CollectionImpl<T>::clear();
}

public count (in element: T): Integer  {
  return super.CollectionImpl<T>::count(element);
}

public equals (in seq: T[0..*] sequence): Boolean {
  return this.size() == seq->size() && this.includesAll(seq);
}

public excludes (in element: T): Boolean {
  return super.CollectionImpl<T>::excludes (element);
}

public excludesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::excludesAll(seq);
}

public includes (in element: T): Boolean {
  return super.CollectionImpl<T>::includes(element);
}

public includesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::includesAll(seq);
}

public isEmpty (): Boolean {
  return super.CollectionImpl<T>::isEmpty();
}

public notEmpty (): Boolean {
  return super.CollectionImpl<T>::notEmpty();
}

public remove (in element: T): Integer {
  return super.CollectionImpl<T>::remove(element);
```

```
  }

  public removeAll (in seq: T[0..*] sequence): Boolean {
    return super.CollectionImpl<T>::removeAll(seq);
  }

  public removeOne (in element: T): Boolean {
    return super.CollectionImpl<T>::removeOne(element);
  }

  public replace (in element: T, in newElement: T): Integer  {
    return super.CollectionImpl<T>::replace(element, newElement);
  }

  public replaceOne (in element: T, in newElement: T): Boolean {
    return super.CollectionImpl<T>::replaceOne(element, newElement);
  }

  public retainAll (in seq: T[0..*] sequence): Boolean {
    return super.CollectionImpl<T>::retainAll(seq);
  }

  public size(): Integer {
    return super.CollectionImpl<T>::size();
  }

  public toSequence(): T[0..*] sequence {
    return this.content;
  }
}
```

B.4.8 CollectionClasses::Impl::List

The `List` class given below implements the abstract `CollectionClasses::List` class. It does this by defining a `content` attribute of the appropriate type with unbounded multiplicity defined as a sequence —that is, a nonunique, ordered list. Operation methods are provided by specializing the `OrderedCollectionImpl` class.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template List class.
*/
class List<T>
  specializes OrderedCollectionImpl<T>, CollectionClasses::List<T> {

  private content: T[0..*] sequence;

  @Create public List (in seq: T[0..*] sequence) {
    super.OrderedCollectionImpl(seq);
  }

  @Destroy public destroy () {
  }

  private setContent(in seq: T[0..*] sequence) {
    this.content = seq;
```

```
}

public add (in element: T): Boolean {
  return super.OrderedCollectionImpl<T>::add(element);
}

public addAt (in index: Integer, in element: T): Boolean {
  return super.OrderedCollectionImpl<T>::addAt(index, element);
}

public addAll (in seq: T[0..*] sequence): Boolean {
  return super.OrderedCollectionImpl<T>::addAll(seq);
}

 public addAllAt (in index: Integer, in seq: T[0..*] sequence): Boolean {
  return super.OrderedCollectionImpl<T>::addAllAt(index, seq);
}

public at (in index: Integer): T[0..1] {
  return super.OrderedCollectionImpl<T>::at(index);
}

public clear () {
  super.OrderedCollectionImpl<T>::clear();
}

public count (in element: T): Integer  {
  return super.OrderedCollectionImpl<T>::count(element);
}

public equals (in seq: T[0..*] sequence): Boolean {
  return super.OrderedCollectionImpl<T>::equals(seq);
}

public first (): T[0..1] {
  return super.OrderedCollectionImpl<T>::first();
}

public indexOf (in element: T) : Integer[0..1] {
  return super.OrderedCollectionImpl<T>::indexOf(element);
}

public last (): T[0..1] {
  return super.OrderedCollectionImpl<T>::last();
}

public excludes (in element: T): Boolean {
  return super.OrderedCollectionImpl<T>::excludes (element);
}

public excludesAll (in seq: T[0..*] sequence): Boolean {
  return super.OrderedCollectionImpl<T>::excludesAll(seq);
}

public includes (in element: T): Boolean {
  return super.OrderedCollectionImpl<T>::includes(element);
}
```

```
  public includesAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::includesAll(seq);
  }

  public isEmpty (): Boolean {
    return super.OrderedCollectionImpl<T>::isEmpty();
  }

  public notEmpty (): Boolean {
    return super.OrderedCollectionImpl<T>::notEmpty();
  }

  public remove (in element: T): Integer {
    return super.OrderedCollectionImpl<T>::remove(element);
  }

  public removeAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::removeAll(seq);
  }

  public removeAt (in index: Integer): T[0..1] {
    return super.OrderedCollectionImpl<T>::removeAt(index);
  }

  public removeOne (in element: T): Boolean {
    return super.OrderedCollectionImpl<T>::removeOne(element);
  }

  public replace (in element: T, in newElement: T): Integer {
    return super.OrderedCollectionImpl<T>::replace(element, newElement);
  }

  public replaceAt (in index: Integer, in element: T): T[0..1] {
    return super.OrderedCollectionImpl<T>::replaceAt(index, element);
  }

  public replaceOne (in element: T, in newElement: T): Boolean {
    return super.OrderedCollectionImpl<T>::replaceOne(element, newElement);
  }

  public retainAll (in seq: T[0..*] sequence): Boolean {
    return super.OrderedCollectionImpl<T>::retainAll(seq);
  }

  public size (): Integer {
    return super.OrderedCollectionImpl<T>::size();
  }

  public subList (in lower: Integer, in upper: Integer): CollectionClasses::List<T>
{
    return new List<T>(this.toSequence()->subsequence(lower, upper));
  }

  public toSequence(): T[0..*] sequence {
    return this.content;
  }
```

```
}
```

B.4.9 CollectionClasses::Impl::Queue

The `List` class given below implements the abstract `CollectionClasses::List` class. It does this by defining a `content` attribute of the appropriate type with unbounded multiplicity defined as a sequence —that is, a nonunique, ordered list. Operation methods are provided by specializing the `CollectionImpl` class. Even though a queue is effectively ordered, it does not provide all the operations defined for other ordered classes and, therefore, does not specialize the `OrderedCollectionImpl` class.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template Queue class.
*/
class Queue<T> specializes CollectionImpl<T>, CollectionClasses::Queue<T> {

  private content: T[0..*] sequence;

  @Create public Queue (in seq: T[0..*] sequence) {
    super.CollectionImpl(seq);
  }

  @Destroy public destroy () {
  }

  protected setContent (in seq: T[0..*] sequence) {
    this.content = seq;
  }

  public add (in element: T): Boolean {
    return super.CollectionImpl<T>::add(element);
  }

  public addAll (in seq: T[0..*] sequence): Boolean {
    return super.CollectionImpl<T>::addAll(seq);
  }

  public addLast (in element : T): Boolean {
    return this.add(element);
  }

  public clear () {
    super.CollectionImpl<T>::clear();
  }

  public count (in element: T): Integer  {
    return super.CollectionImpl<T>::count(element);
  }

  public equals (in seq: T[0..*] sequence): Boolean {
    return this.size() == seq->size() && this.includesAll(seq);
  }

  public excludes (in element: T): Boolean {
    return super.CollectionImpl<T>::excludes (element);
  }
```

```
public excludesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::excludesAll(seq);
}

public first(): T[0..1] {
  return this.toSequence()->first();
}

public includes (in element: T): Boolean {
  return super.CollectionImpl<T>::includes(element);
}

public includesAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::includesAll(seq);
}

public isEmpty (): Boolean {
  return super.CollectionImpl<T>::isEmpty();
}

public notEmpty (): Boolean {
  return super.CollectionImpl<T>::notEmpty();
}

public remove (in element: T): Integer {
  return super.CollectionImpl<T>::remove(element);
}

public removeAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::removeAll(seq);
}

public removeFirst (): T[0..1] {
  result = this.toSequence()->first();
  this.setContent(this.toSequence()->subsequence(2,this.size()));
  return result;
}

public removeFirstOne (in element: T): T[0..1] {
  return this.removeOne(element)? element: null;
}

public removeOne (in element: T): Boolean {
  return super.CollectionImpl<T>::removeOne(element);
}

public replace (in element: T, in newElement: T): Integer  {
  return super.CollectionImpl<T>::replace(element, newElement);
}

public replaceOne (in element: T, in newElement: T): Boolean {
  return super.CollectionImpl<T>::replaceOne(element, newElement);
}

public retainAll (in seq: T[0..*] sequence): Boolean {
  return super.CollectionImpl<T>::retainAll(seq);
```

```
  }

  public size(): Integer {
    return super.CollectionImpl<T>::size();
  }

  public toSequence (): T[0..*] sequence {
    return this.content;
  }

}
```

B.4.10 CollectionClasses::Impl::Deque

The `Deque` class given below implements the abstract `CollectionClasses::Deque` class. It does this by extending the `Queue` implementation class with the additional operations defined for a `Deque`.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template Deque class.
*/
class Deque<T> specializes Queue<T>, CollectionClasses::Deque<T> {

  @Create public Deque (in seq: T[0..*] sequence) {
    super.Queue<T>::Queue(seq);
  }

  @Destroy public destroy () {
    super.Queue<T>::destroy();
  }

  @Create private Queue (in seq: T[0..*] sequence) {
    this.Deque(seq);
  }

  public add (in element: T): Boolean {
    return super.Queue<T>::add(element);
  }

  public addAll (in seq: T[0..*] sequence): Boolean {
    return super.Queue<T>::addAll(seq);
  }

  public addFirst (in element: T): Boolean {
    this.setContent(this.toSequence()->includeAt(1, element));
    return true;
  }

  public addLast (in element : T): Boolean {
    return this.add(element);
  }

  public clear () {
    super.Queue<T>::clear();
  }
```

```
public count (in element: T): Integer  {
  return super.Queue<T>::count(element);
}

public equals (in seq: T[0..*] sequence): Boolean {
  return this.size() == seq->size() && this.includesAll(seq);
}

public excludes (in element: T): Boolean {
  return super.Queue<T>::excludes (element);
}

public excludesAll (in seq: T[0..*] sequence): Boolean {
  return super.Queue<T>::excludesAll(seq);
}

public first(): T[0..1] {
  return this.toSequence()->first();
}

public includes (in element: T): Boolean {
  return super.Queue<T>::includes(element);
}

public includesAll (in seq: T[0..*] sequence): Boolean {
  return super.Queue<T>::includesAll(seq);
}

public isEmpty (): Boolean {
  return super.Queue<T>::isEmpty();
}

public last (): T[0..1] {
  return this.toSequence()->last();
}

public notEmpty (): Boolean {
  return super.Queue<T>::notEmpty();
}

public remove (in element: T): Integer {
  return super.Queue<T>::remove(element);
}

public removeAll (in seq: T[0..*] sequence): Boolean {
  return super.Queue<T>::removeAll(seq);
}

public removeFirst (): T[0..1] {
  result = this.toSequence()->first();
  this.setContent(this.toSequence()->subsequence(2,this.size()));
  return result;
}

public removeFirstOne (in element: T): T[0..1] {
  return this.removeOne(element)? element: null;
}
```

```
  public removeLast (): T[0..1] {
    result = this.last();
    this.setContent(this.toSequence()->subsequence(1,this.size()-1));
    return result;
  }

  public removeLastOne (in element: T): T[0..1] {
    n = this.size();
    for (i in 1..n) {
      e = this.toSequence()->at(n - i + 1);
      if (e == element) {
        this.setContent(this.toSequence()->excludeAt(i));
        return e;
      }
    }
    return null;
  }

  public removeOne (in element: T): Boolean {
    return super.Queue<T>::removeOne(element);
  }

  public replace (in element: T, in newElement: T): Integer  {
    return super.Queue<T>::replace(element, newElement);
  }

  public replaceOne (in element: T, in newElement: T): Boolean {
    return super.Queue<T>::replaceOne(element, newElement);
  }

  public retainAll (in seq: T[0..*] sequence): Boolean {
    return super.Queue<T>::retainAll(seq);
  }

  public size(): Integer {
    return super.Queue<T>::size();
  }

  public toSequence (): T[0..*] sequence {
    return super.Queue<T>::toSequence();
  }

}
```

B.4.11 CollectionClasses::Impl::Map

The `Map` class given below implements the abstract `CollectionClasses::List` Map. It does this by storing a set of entries in an attribute. Lookup of entries is implemented in the private `indexOf` operation.

```
namespace Alf::Library::CollectionClasses::Impl;
private import Alf::Library::CollectionFunctions::*;
/**
The concrete implementation of the standard library template Map class.
*/
class Map<Key, Value> specializes CollectionClasses::Map<Key,Value> {
```

```
private entries: Entry[0..*];

@Create public Map (in entries: Entry[0..*]) {
  this.putAll(entries);
}

@Destroy public destroy () {
}

private indexOf(in key: Key): Integer[0..1] {
  return this.entries.key->indexOf(key);
}

public entries (): CollectionClasses::Set<Entry> {
  return new Set<Entry>(this.entries);
}

public clear () {
  this.entries = null;
}

public excludesAll (in entries: Entry[0..*]): Boolean {
  return this.entries->excludesAll(entries);
}

public get (in key: Key): Value[0..1] {
  return this.entries->select e (e.key == key)[1].value;
}

public includesAll (in entries: Entry[0..*]): Boolean {
  return this.entries->includesAll(entries);
}

public includesKey (in key: Key): Boolean {
  return this.entries.key->includes(key);
}

public includesValue (in value: Value[0..1]): Boolean {
  return this.entries->exists e (e.value == value);
}

public isEmpty (): Boolean {
  return this.entries->isEmpty();
}

public keys (): CollectionClasses::Set<Key> {
  return new Set<Key>(this.entries.key);
}

public notEmpty (): Boolean {
  return this.entries->notEmpty();
}

public put (in key: Key, in value: Value[0..1]): Value[0..1] {
    result = this.remove(key);
    this.entries->add(new Entry(key,value));
```

```
      return result;
  }

  public putAll (in entries: Entry[0..*]) {
    entries->iterate e (this.put(e.key, e.value));
  }

  public remove (in key: Key): Value[0..1] {
    result = this.get(key);
    this.entries = this.entries->reject e (e.key == key);
    return result;
  }

  public removeAll (in keys: Key[0..*]) {
    keys->iterate k (this.remove(k));
  }

  public size (): Integer {
    this.entries->size();
  }

  public toSequence (): Entry[0..*] sequence {
    return this.entries;
  }

  public values (): CollectionClasses::Bag<Value> {
    return new Bag<Value>(this.entries.value);
  }
}
```

Annex C: Consolidated LL Grammar

(informative)

C.1 Introduction

The grammar productions presented for Alf in the main body of this document are intended to describe the lexical structure and syntax of the language in a way that is as clear as possible to a reader of the specification. However, the resulting grammar is not necessarily appropriate as the basis for automated parsing of the language. The grammar presented in this annex is equivalent to the normative grammar for Alf, but is specifically intended for use in automated parsing.

NOTE. This grammar has been implemented using JavaCC parser generator (https://javacc.dev.java.net/). The format of the lexical and syntactic productions in this annex is therefore as required for this tool.

C.2 Lexical Analyzer

The lexical analyzer is defined as a state machine that generates a sequence of tokens that are fed to the parser. The operation of this state machine is given by the set of productions listed below. Each production has the following form:

- A list of names of the states in which the production is active. The analyzer begins in the `DEFAULT` state.
- The action of the production, which is one of the following:
  - `SKIP` – Any text matched by the production is skipped and not passed to the parser.
  - `TOKEN` – Text matched by the production forms the image of a token that is passed to the parser.
  - `MORE` – Text matched by the production is buffered until the next match of a production with a `SKIP` or `TOKEN` action, at which time all buffered text is either skipped or used in the construction of a token respectively.
- A list of alternatives, each of which consists of an optional name, a *regular expression* and an optional *target state.* If the regular expression in an alternative matches the input text at the current position, then this text is consumed, the production action is carried out and the analyzer moves to the target state. If no target state is given, then the analyzer remains in the same state. If a name is given in a matching alternative and the production action is to produce a token, then the token is given that name.

Productions are checked in order, so earlier productions have precedence over later productions.

```
/* WHITE SPACE */

<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> SKIP : {
  " "
| "\t"
| "\f"
}

<DEFAULT> SKIP : {
  "\n"
```

```
| "\r"
}

/* STATEMENT ANNOTATIONS */

<DEFAULT> TOKEN : {
  <SLASH_SLASH_AT: "//@"> : IN_STATEMENT_ANNOTATION
| <SLASH_STAR_AT: "/*@"> : IN_IN_LINE_ANNOTATION
}

<IN_STATEMENT_ANNOTATION> TOKEN : {
  <EOL: ("//" (~["\n","\r"])*)? ("\n" | "\r" ("\n")?)> : DEFAULT
}

<IN_IN_LINE_ANNOTATION> SKIP : {
  <("//" (~["\n","\r"])*)? ("\n" | "\r" ("\n")?)> : IN_DOCUMENTATION_COMMENT
}

/* COMMENTS */

<DEFAULT> SKIP : {
  <"/**" ~["/"]> : IN_DOCUMENTATION_COMMENT
}

<DEFAULT> MORE : {
  "//" : IN_END_OF_LINE_COMMENT
| "/*" : IN_IN_LINE_COMMENT
}

<IN_END_OF_LINE_COMMENT> SKIP : {
  <END_OF_LINE_COMMENT: "\n" | "\r" ("\n")?> : DEFAULT
}

<IN_END_OF_LINE_COMMENT> MORE : {
  <~[]>
}

<IN_IN_LINE_COMMENT> MORE : {
  <~["*"]>
| "*" : IN_IN_LINE_COMMENT_STAR
}

<IN_IN_LINE_COMMENT_STAR> MORE : {
  <~["/"]> : IN_IN_LINE_COMMENT
}

<IN_IN_LINE_COMMENT_STAR> SKIP : {
  <IN_LINE_COMMENT: "/"> : DEFAULT
}

<IN_DOCUMENTATION_COMMENT> MORE : {
  <~["*"]>
| "*" : IN_DOCUMENTATION_COMMENT_STAR
}

<IN_DOCUMENTATION_COMMENT_STAR> TOKEN : {
  <DOCUMENTATION_COMMENT: "/"> : DEFAULT
```

```
}

<IN_DOCUMENTATION_COMMENT_STAR> MORE : {
  <~["/"]> : IN_DOCUMENTATION_COMMENT
}

/* RESERVED WORDS */

<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> TOKEN : {
  <ABSTRACT: "abstract">
| <ACCEPT: "accept">
| <ACTIVE: "active">
| <ACTIVITY: "activity">
| <ALL_INSTANCES: "allInstances">
| <ANY: "any">
| <AS: "as">
| <ASSOC: "assoc">
| <BREAK: "break">
| <CASE: "case">
| <CLASS: "class">
| <CLASSIFY: "classify">
| <CLEAR_ASSOC: "clearAssoc">
| <COMPOSE: "compose">
| <CREATE_LINK: "createLink">
| <DATATYPE: "datatype">
| <DEFAULT_: "default">
| <DESTROY_LINK: "destroyLink">
| <DO: "do">
| <ELSE: "else">
| <ENUM: "enum">
| <FOR: "for">
| <FROM: "from">
| <HASTYPE: "hastype">
| <IF: "if">
| <IMPORT: "import">
| <IN: "in">
| <INOUT: "inout">
| <INSTANCEOF: "instanceof">
| <LET: "let">
| <NAMESPACE: "namespace">
| <NEW: "new">
| <NONUNIQUE: "nonunique">
| <NULL: "null">
| <OR: "or">
| <ORDERED: "ordered">
| <OUT: "out">
| <PACKAGE: "package">
| <PRIVATE: "private">
| <PROTECTED: "protected">
| <PUBLIC: "public">
| <RECEIVE: "receive">
| <REDEFINES: "redefines">
| <REDUCE: "reduce">
| <RETURN: "return">
| <SEQUENCE: "sequence">
| <SPECIALIZES: "specializes">
| <SUPER: "super">
```

```
| <SIGNAL: "signal">
| <SWITCH: "switch">
| <THIS: "this">
| <TO: "to">
| <WHILE: "while">
}

/* NAMES */

<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> TOKEN : {
  <IDENTIFIER: <IDENTIFIER_LETTER> (<IDENTIFIER_LETTER_OR_DIGIT>)*>
| <#IDENTIFIER_LETTER_OR_DIGIT: <IDENTIFIER_LETTER> | <DIGIT>>
| <#IDENTIFIER_LETTER: ["a"-"z","A"-"Z","_"]>
| <#DIGIT: "0" | <NONZERO_DIGIT>>
| <#NONZERO_DIGIT: ["1"-"9"]>
| <UNRESTRICTED_NAME: "\'" (<NAME_CHARACTER>)+ "\'">
| <#NAME_CHARACTER: ~["\'","\\"] | <ESCAPE_CHARACTER>>
| <#ESCAPE_CHARACTER: "\\" <ESCAPED_CHARACTER>>
| <#ESCAPED_CHARACTER: ["\'","\"","b","f","n","\\"]>
}

/* PRIMITIVE LITERALS */

<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> TOKEN : {
  <BOOLEAN_LITERAL: "true" | "false">
| <BINARY_LITERAL: ("0b" | "0B") <BINARY_DIGIT> (("_")? <BINARY_DIGIT>)*>
| <#BINARY_DIGIT: ["0","1"]>
| <HEX_LITERAL: ("0x" | "0X") <HEX_DIGIT> (("_")? <HEX_DIGIT>)*>
| <#HEX_DIGIT: <DIGIT> | ["a"-"f","A"-"F"]>
| <OCTAL_LITERAL: "0" (("_")? <OCTAL_DIGIT>)+>
| <#OCTAL_DIGIT: ["0"-"7"]>
| <DECIMAL_LITERAL: "0" | <NONZERO_DIGIT> (("_")? <DIGIT>)*>
| <STRING_LITERAL: "\"" (<STRING_CHARACTER>)* "\"">
| <#STRING_CHARACTER: ~["\"","\\"]>
}

/* PUNCTUATION */

<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> TOKEN : {
  <LPAREN: "(">
| <RPAREN: ")">
| <LBRACE: "{">
| <RBRACE: "}">
| <LBRACKET: "[">
| <RBRACKET: "]">
| <SEMICOLON: ";">
| <COMMA: ",">
| <DOT: ".">
| <DOUBLE_DOT: "..">
| <COLON: ":">
| <DOUBLE_COLON: "::">
| <ARROW: "->">
| <THICK_ARROW: "=>">
}

/* OPERATORS */
```

```
<DEFAULT,IN_STATEMENT_ANNOTATION,IN_IN_LINE_ANNOTATION> TOKEN : {
  <ASSIGN: "=">
| <GT: ">">
| <LT: "<">
| <BANG: "!">
| <TILDE: "~">
| <HOOK: "?">
| <DOUBLE_HOOK: "??">
| <AT: "@">
| <DOLLAR: "$">
| <EQ: "==">
| <LE: "<=">
| <GE: ">=">
| <NE: "!=">
| <SC_OR: "||">
| <SC_AND: "&&">
| <INCR: "++">
| <DECR: "--">
| <PLUS: "+">
| <MINUS: "-">
| <STAR: "*">
| <SLASH: "/">
| <LOGICAL_AND: "&">
| <LOGICAL_OR: "|">
| <XOR: "^">
| <REM: "%">
| <LSHIFT: "<<">
| <RSHIFT: ">>">
| <URSHIFT: ">>>">
| <PLUSASSIGN: "+=">
| <MINUSASSIGN: "-=">
| <STARASSIGN: "*=">
| <SLASHASSIGN: "/=">
| <ANDASSIGN: "&=">
| <ORASSIGN: "|=">
| <XORASSIGN: "^=">
| <REMASSIGN: "%=">
| <LSHIFTASSIGN: "<<=">
| <RSHIFTASSIGN: ">>=">
| <URSHIFTASSIGN: ">>>=">
}
```

C.3 Parser

The syntax grammar given below is an LL grammar, that is, a grammar designed for “top down” parsing (such as recursive descent). A lookahead of only one token is required for most of the grammar. In all other cases except one, a maximum lookahead of three tokens is required. The parsing of qualified names with template bindings, however, requires a potentially unbounded lookahead to distinguish the use of a “<” as an opening bracket of a template binding from its use as a less-than sign.

As discussed in 8.2, the parsing of a syntactic element with the form of a qualified name using the dot notation may be ambiguous. In such cases, the grammar below always parses the potentially ambiguous element as a qualified name (using the non-terminal “`PotentiallyAmbiguousQualifiedName`”). If such an element is later disambiguated as a feature reference, this will also generally require reparsing of

the expression containing the ambiguous qualified name (e.g., a name expression becomes a property access expression and a behavior invocation becomes a feature invocation).

Finally, the LL form of the grammar given here is considerably simplified by allowing any unary expression as the left-hand side of an assignment and any primary expression as the operand of an increment or decrement expression, even though the required form for a left-hand side is actually more restrictive (see 8.8). This requires that the left-hand side of an assignment and the operand of an increment or decrement expression be checked after parsing to ensure that it has the form of a name expression, a property access expression or a sequence access expression whose primary expression is a name expression or a property access expression. Note that a similar static check needs to be performed anyway on the argument expressions for inout parameters (see 8.3.8).

NOTE. The grammar as presented below uses a slightly different EBNF notation than in the main text, with `(…)?` meaning optional, `(…)*` meaning zero or more and `(..)+` meaning one or more. Terminal symbols are written in the form `<TOKEN_NAME>`, were `TOKEN_NAME` is the name of a lexical token, as given by a lexical analyzer production above.

```
/***************
 *    NAMES    *
 ***************/
Name                          = <IDENTIFIER> | <UNRESTRICTED_NAME>
QualifiedName                 = UnqualifiedName
                                ( ColonQualifiedNameCompletion
                                | DotQualifiedNameCompletion
                                )?
PotentiallyAmbiguousQualifiedName
                              = UnqualifiedName
                                ( ColonQualifiedNameCompletion
                                | DotQualifiedNameCompletion /* AMBIGUOUS */
                                )?
ColonQualifiedName            = UnqualifiedName ColonQualifiedNameCompletion
ColonQualifiedNameCompletion  = ( <DOUBLE_COLON> NameBinding )+
DotQualifiedName              = UnqualifiedName DotQualifiedNameCompletion
DotQualifiedNameCompletion    = ( <DOT> NameBinding )+
UnqualifiedName               = NameBinding
NameBinding                   = Name ( TemplateBinding )?
                                     /* ^ Unbounded lookahead required here */
TemplateBinding               = <LT> ( NamedTemplateBinding
                                      | PositionalTemplateBinding ) <GT>
PositionalTemplateBinding     = QualifiedName ( <COMMA> QualifiedName )*
NamedTemplateBinding          = TemplateParameterSubstitution
                                ( <COMMA> TemplateParameterSubstitution )*
TemplateParameterSubstitution = Name <THICK_ARROW> QualifiedName

/***************
 * EXPRESSIONS *
 ***************/
Expression                    = UnaryExpression ExpressionCompletion
NonNameExpression             = NonNameUnaryExpression ExpressionCompletion
NameToExpressionCompletion    = ( NameToPrimaryExpression )?
                                PrimaryToExpressionCompletion
PrimaryToExpressionCompletion = PostfixExpressionCompletion
                                ExpressionCompletion
ExpressionCompletion          = AssignmentExpressionCompletion
                              | ConditionalExpressionCompletion

/* PRIMARY EXPRESSIONS */
PrimaryExpression             = ( NameOrPrimaryExpression
                                | BaseExpression
                                | ParenthesizedExpression )
                                PrimaryExpressionCompletion
BaseExpression                = LiteralExpression
                              | ThisExpression
                              | SuperInvocationExpression
                              | InstanceCreationOrSequenceConstructionExpression
                              | SequenceAnyExpression
NameToPrimaryExpression       = <DOT> ( LinkOperationCompletion
                                      | ClassExtentExpressionCompletion )
                              | SequenceConstructionExpressionCompletion
                              | BehaviorInvocation
PrimaryExpressionCompletion   = ( Feature ( FeatureInvocation )?
                                | SequenceOperationOrReductionOrExpansion
                                | Index
                                )*
```

```
/* LITERAL EXPRESSIONS */
LiteralExpression              = <BOOLEAN_LITERAL>
                               | <DECIMAL_LITERAL>
                               | <BINARY_LITERAL>
                               | <OCTAL_LITERAL>
                               | <HEX_LITERAL>
                               | <STRING_LITERAL>
                               | <STAR>

/* NAME EXPRESSIONS */
NameOrPrimaryExpression        = PotentiallyAmbiguousQualifiedName
                                 ( NameToPrimaryExpression )?

/* THIS EXPRESSIONS */
ThisExpression                 = <THIS> ( Tuple )?

/* PARENTHESIZED EXPRESSIONS */
ParenthesizedExpression        = <LPAREN> Expression <RPAREN>

/* PROPERTY ACCESS EXPRESSIONS */
Feature                        = <DOT> NameBinding

/* INVOCATION EXPRESSIONS */
Tuple                          = <LPAREN>
                                 ( NamedTupleExpressionList
                                 | ( PositionalTupleExpressionList )?
                                 ) <RPAREN>
PositionalTupleExpressionList = Expression
                                 PositionalTupleExpressionListCompletion
PositionalTupleExpressionListCompletion
                               = ( <COMMA> Expression )*
NamedTupleExpressionList       = NamedExpression ( <COMMA> NamedExpression )*
NamedExpression                = Name <THICK_ARROW> Expression
BehaviorInvocation             = Tuple
FeatureInvocation              = Tuple
SuperInvocationExpression      = <SUPER> ( <DOT> QualifiedName )? Tuple

/* INSTANCE CREATION EXPRESSIONS */
InstanceCreationOrSequenceConstructionExpression
                               = <NEW> QualifiedName
                                 ( SequenceConstructionExpressionCompletion
                                 | Tuple )

/* LINK OPERATION EXPRESSIONS */
LinkOperationCompletion        = LinkOperation LinkOperationTuple
LinkOperation                  = <CREATE_LINK>
                               | <DESTROY_LINK>
                               | <CLEAR_ASSOC>
```

```
LinkOperationTuple            = <LPAREN>
                                ( Name
                                  ( Index
                                    ( <THICK_ARROW>
                                      IndexedNamedExpressionListCompletion
                                    | PrimaryToExpressionCompletion
                                      PositionalTupleExpressionListCompletion
                                    )
                                  | <THICK_ARROW>
                                    IndexedNamedExpressionListCompletion
                                  )
                                | PositionalTupleExpressionList
                                )? <RPAREN>
IndexedNamedExpressionListCompletion
                              = Expression
                                ( <COMMA> IndexedNamedExpression )*
IndexedNamedExpression        = Name ( Index )? <THICK_ARROW> Expression

/* CLASS EXTENT EXPRESSIONS */
ClassExtentExpressionCompletion
                              = <ALL_INSTANCES> <LPAREN> <RPAREN>

/* SEQUENCE CONSTRUCTION EXPRESSIONS */
SequenceAnyExpression         = <ANY>
                                SequenceConstructionExpressionCompletion
                              | <NULL>
SequenceConstructionExpressionCompletion
                              = ( MultiplicityIndicator )? <LBRACE>
                                ( SequenceElements )? <RBRACE>
MultiplicityIndicator         = <LBRACKET> <RBRACKET>
SequenceElements              = Expression ( <DOUBLE_DOT> Expression
                                           | SequenceElementListCompletion )
                              | SequenceInitializationExpression
                                SequenceElementListCompletion
SequenceElementListCompletion = ( <COMMA> SequenceElement )* ( <COMMA> )?
SequenceElement               = Expression
                              | SequenceInitializationExpression
SequenceInitializationExpression
                              = ( <NEW> )? <LBRACE> SequenceElements <RBRACE>

/* SEQUENCE ACCESS EXPRESSIONS */
Index                         = <LBRACKET> Expression <RBRACKET>

/* SEQUENCE OPERATION, REDUCTION AND EXPANSION EXPRESSIONS */
SequenceOperationOrReductionOrExpansion
                              = <ARROW>
                                ( QualifiedName Tuple
                                | <REDUCE> ( <ORDERED> )? QualifiedName
                                | <IDENTIFIER> Name
                                  <LPAREN> Expression <RPAREN>
                                )

/* INCREMENT OR DECREMENT EXPRESSIONS */
PostfixExpressionCompletion   = PrimaryExpressionCompletion
                                ( PostfixOperation )?
PostfixOperation              = AffixOperator
PrefixExpression              = AffixOperator PrimaryExpression
```

```
AffixOperator                 = ( <INCR> | <DECR> )

/* UNARY EXPRESSIONS */
UnaryExpression               = PostfixOrCastExpression
                              | NonPostfixNonCastUnaryExpression
PostfixOrCastExpression       = NonNamePostfixOrCastExpression
                              | NameOrPrimaryExpression
                                PostfixExpressionCompletion
NonNameUnaryExpression        = NonNamePostfixOrCastExpression
                              | NonPostfixNonCastUnaryExpression
NonNamePostfixOrCastExpression
                              = <LPAREN>
                                ( <ANY> <RPAREN> CastCompletion
                                | PotentiallyAmbiguousQualifiedName
                                  ( <RPAREN> CastCompletion
                                  | NameToExpressionCompletion <RPAREN>
                                    PostfixExpressionCompletion
                                  )
                                | NonNameExpression <RPAREN>
                                  PostfixExpressionCompletion
                                )
                              | BaseExpression PostfixExpressionCompletion
NonPostfixNonCastUnaryExpression
                              = PrefixExpression
                              | NumericUnaryExpression
                              | BooleanNegationExpression
                              | BitStringComplementExpression
                              | IsolationExpression
BooleanNegationExpression     = <BANG> UnaryExpression
BitStringComplementExpression = <TILDE> UnaryExpression
NumericUnaryExpression        = NumericUnaryOperator UnaryExpression
NumericUnaryOperator          = <PLUS> | <MINUS>
IsolationExpression           = <DOLLAR> UnaryExpression
CastExpression                = <LPAREN> TypeName <RPAREN> CastCompletion
CastCompletion                = PostfixOrCastExpression
                              | BooleanNegationExpression
                              | BitStringComplementExpression
                              | IsolationExpression

/* ARITHMETIC EXPRESSIONS */
MultiplicativeExpression      = UnaryExpression
                                MultiplicativeExpressionCompletion
MultiplicativeExpressionCompletion
                              = ( MultiplicativeOperator UnaryExpression )*
MultiplicativeOperator        = <STAR> | <SLASH> | <REM>
AdditiveExpression            = UnaryExpression AdditiveExpressionCompletion
AdditiveExpressionCompletion  = MultiplicativeExpressionCompletion
                                ( AdditiveOperator MultiplicativeExpression)*
AdditiveOperator              = ( <PLUS> | <MINUS> )

/* SHIFT EXPRESSIONS */
ShiftExpression               = UnaryExpression ShiftExpressionCompletion
ShiftExpressionCompletion     = AdditiveExpressionCompletion
                                ( ShiftOperator AdditiveExpression )*
ShiftOperator                 = <LSHIFT> | <RSHIFT> | <URSHIFT>

/* RELATIONAL EXPRESSIONS */
```

```
RelationalExpression          = UnaryExpression
                                RelationalExpressionCompletion
RelationalExpressionCompletion
                              = ShiftExpressionCompletion
                                ( RelationalOperator ShiftExpression )?
RelationalOperator            = <LT> | <GT> | <LE> | <GE>

/* CLASSIFICATION EXPRESSIONS */
ClassificationExpression      = UnaryExpression
                                ClassificationExpressionCompletion
ClassificationExpressionCompletion
                              = RelationalExpressionCompletion
                              ( ClassificationOperator QualifiedName )?
ClassificationOperator        = <INSTANCEOF> | <HASTYPE>

/* EQUALITY EXPRESSIONS */
EqualityExpression            = UnaryExpression
                                ClassificationExpressionCompletion
EqualityExpressionCompletion  = ClassificationExpressionCompletion
                                ( EqualityOperator ClassificationExpression)*
EqualityOperator              = <EQ> | <NE>

/* LOGICAL EXPRESSIONS */
AndExpression                 = UnaryExpression AndExpressionCompletion
AndExpressionCompletion       = EqualityExpressionCompletion
                                ( <LOGICAL_AND> EqualityExpression )*
ExclusiveOrExpression         = UnaryExpression
                                ExclusiveOrExpressionCompletion
ExclusiveOrExpressionCompletion
                              = AndExpressionCompletion
                                ( <XOR> AndExpression )*
InclusiveOrExpression         = UnaryExpression
                                InclusiveOrExpressionCompletion
InclusiveOrExpressionCompletion
                              = ExclusiveOrExpressionCompletion
                                ( <LOGICAL_OR> ExclusiveOrExpression )*

/* CONDITIONAL LOGICAL EXPRESSIONS */
ConditionalAndExpression      = UnaryExpression
                                ConditionalAndExpressionCompletion
ConditionalAndExpressionCompletion
                              = InclusiveOrExpressionCompletion
                                ( <SC_AND> InclusiveOrExpression )*
ConditionalOrExpression       = UnaryExpression
                                ConditionalOrExpressionCompletion
ConditionalOrExpressionCompletion
                              = ConditionalAndExpressionCompletion
                                ( <SC_OR> ConditionalAndExpression )*

/* NULL-COALESCING EXPRESSIONS */
NullCoalescingExpression      = UnaryExpression
                                NullCoalescingExpressionCompletion
NullCoalescingExpressionCompletion
                              = ConditionalOrExpressionCompletion
                                ( <DOUBLE_HOOK> NullCoalescingExpression )?

/* CONDITIONAL-TEST EXPRESSIONS */
```

```
ConditionalExpression         = UnaryExpression
                                NullCoalescingExpressionCompletion
ConditionalExpressionCompletion
                              = NullCoalescingExpressionCompletion
                                ( <HOOK> Expression <COLON>
                                  ConditionalExpression )?

/* ASSIGNMENT EXPRESSIONS */
AssignmentExpressionCompletion
                              = AssignmentOperator Expression
AssignmentOperator            = <ASSIGN>
                              | <PLUSASSIGN>
                              | <MINUSASSIGN>
                              | <STARASSIGN>
                              | <SLASHASSIGN>
                              | <REMASSIGN>
                              | <ANDASSIGN>
                              | <ORASSIGN>
                              | <XORASSIGN>
                              | <LSHIFTASSIGN>
                              | <RSHIFTASSIGN>
                              | <URSHIFTASSIGN>

/**************
 * STATEMENTS *
 **************/
StatementSequence             = ( DocumentedStatement )*
DocumentedStatement           = ( <DOCUMENTATION_COMMENT> )? Statement
Statement                     = AnnotatedStatement
                              | InLineStatement
                              | BlockStatement
                              | EmptyStatement
                              | LocalNameDeclarationOrExpressionStatement
                              | LocalNameDeclarationStatement
                              | IfStatement
                              | SwitchStatement
                              | WhileStatement
                              | ForStatement
                              | DoStatement
                              | BreakStatement
                              | ReturnStatement
                              | AcceptStatement
                              | ClassifyStatement

/* BLOCK */
Block                         = <LBRACE> StatementSequence <RBRACE>

/* ANNOTATED STATEMENTS */
AnnotatedStatement            = <SLASH_SLASH_AT> Annotations <EOL> Statement
Annotations                   = Annotation ( <AT> Annotation )*
Annotation                    = <IDENTIFIER> ( <LPAREN> NameList <RPAREN> )?
NameList                      = Name ( <COMMA> Name )*

/* IN-LINE STATEMENTS */
InLineStatement               = <SLASH_STAR_AT> <IDENTIFIER> <LPAREN> Name
                                <RPAREN> <DOCUMENTATION_COMMENT>
```

```
/* BLOCK STATEMENTS */
BlockStatement                = Block

/* EMPTY STATEMENTS */
EmptyStatement                = <SEMICOLON>

/* LOCAL NAME DECLARATION AND EXPRESSION STATEMENTS */
LocalNameDeclarationOrExpressionStatement
                              = PotentiallyAmbiguousQualifiedName
                                ( ( MultiplicityIndicator )? Name
                                  LocalNameDeclarationStatementCompletion
                                | NameToExpressionCompletion <SEMICOLON>
                                )
                              | NonNameExpression <SEMICOLON>
LocalNameDeclarationStatement = <LET> Name <COLON> TypeName
                                ( MultiplicityIndicator )?
                                LocalNameDeclarationStatementCompletion
LocalNameDeclarationStatementCompletion
                              = <ASSIGN> InitializationExpression <SEMICOLON>
InitializationExpression      = Expression
                              | SequenceInitializationExpression
                              | InstanceInitializationExpression
InstanceInitializationExpression
                              = <NEW> Tuple

/* IF STATEMENTS */
IfStatement                   = <IF> SequentialClauses ( FinalClause )?
SequentialClauses             = ConcurrentClauses
                                ( <ELSE> <IF> ConcurrentClauses )*
ConcurrentClauses             = NonFinalClause ( <OR> <IF> NonFinalClause )*
NonFinalClause                = <LPAREN> Expression <RPAREN> Block
FinalClause                   = <ELSE> Block

/* SWITCH STATEMENTS */
SwitchStatement               = <SWITCH> <LPAREN> Expression <RPAREN>
                                <LBRACE> ( SwitchClause )*
                                ( SwitchDefaultClause )? <RBRACE>
SwitchClause                  = SwitchCase ( SwitchCase )*
                                NonEmptyStatementSequence
SwitchCase                    = <CASE> Expression <COLON>
SwitchDefaultClause           = <DEFAULT_> <COLON> NonEmptyStatementSequence
NonEmptyStatementSequence     = ( DocumentedStatement )+

/* WHILE STATEMENTS */
WhileStatement                = <WHILE> <LPAREN> Expression <RPAREN> Block

/* DO STATEMENTS */
DoStatement                   = <DO> Block <WHILE> <LPAREN> Expression
                                <RPAREN> <SEMICOLON>

/* FOR STATEMENTS */
ForStatement                  = <FOR> <LPAREN> ForControl <RPAREN> Block
ForControl                    = LoopVariableDefinition
                                ( <COMMA> LoopVariableDefinition )*
LoopVariableDefinition        = Name <IN> Expression
                                ( <DOUBLE_DOT> Expression )?
                              | QualifiedName Name <COLON> Expression
```

```
/* BREAK STATEMENTS */
BreakStatement                = <BREAK> <SEMICOLON>

/* RETURN STATEMENTS */
ReturnStatement               = <RETURN> ( Expression )? <SEMICOLON>

/* ACCEPT STATEMENTS */
AcceptStatement               = AcceptClause
                                ( SimpleAcceptStatementCompletion
                                | CompoundAcceptStatementCompletion )
SimpleAcceptStatementCompletion
                              = <SEMICOLON>
CompoundAcceptStatementCompletion
                              = Block ( <OR> AcceptBlock )*
AcceptBlock                   = AcceptClause Block
AcceptClause                  = <ACCEPT> <LPAREN> ( Name <COLON> )?
                                QualifiedNameList <RPAREN>

/* CLASSIFY STATEMENTS */
ClassifyStatement             = <CLASSIFY> Expression ClassificationClause
                                <SEMICOLON>
ClassificationClause          = ClassificationFromClause
                                ( ClassificationToClause )?
                              | ( ReclassifyAllClause )?
                                ClassificationToClause
ClassificationFromClause      = <FROM> QualifiedNameList
ClassificationToClause        = <TO> QualifiedNameList
ReclassifyAllClause           = <FROM> <STAR>
QualifiedNameList             = QualifiedName ( <COMMA> QualifiedName )*

/*********
 * UNITS *
 *********/
UnitDefinition                = ( NamespaceDeclaration )?
                                ( ImportDeclaration )*
                                ( <DOCUMENTATION_COMMENT> )?
                                StereotypeAnnotations NamespaceDefinition
StereotypeAnnotations         = ( StereotypeAnnotation )*
StereotypeAnnotation          = <AT> QualifiedName
                                ( <LPAREN> TaggedValues <RPAREN> )?
TaggedValues                  = QualifiedNameList
                              | TaggedValueList
TaggedValueList               = TaggedValue ( "," TaggedValue )*
TaggedValue                   = Name <THICK_ARROW>
                                ( <BOOLEAN_LITERAL>
                                | <STRING_LITERAL>
                                | <STAR>
                                | ( <PLUS> | <MINUS> )?
                                  ( <DECIMAL_LITERAL>
                                  | <BINARY_LITERAL>
                                  | <OCTAL_LITERAL>
                                  | <HEX_LITERAL>
                                  )
                                )
NamespaceDeclaration          = <NAMESPACE> QualifiedName <SEMICOLON>
```

```
ImportDeclaration             = ImportVisibilityIndicator <IMPORT>
                                ImportReference <SEMICOLON>
ImportVisibilityIndicator     = <PUBLIC> | <PRIVATE>
ImportReference               = ColonQualifiedName
                                ( <DOUBLE_COLON> <STAR>
                                | AliasDefinition )?
                              | DotQualifiedName
                                ( <DOT> <STAR> | AliasDefinition )?
                              | Name
                                ( ( <DOUBLE_COLON> | <DOT> ) <STAR>
                                | AliasDefinition )?
AliasDefinition               = <AS> Name

/* NAMESPACES */
NamespaceDefinition           = PackageDefinition | ClassifierDefinition
VisibilityIndicator           = ImportVisibilityIndicator | <PROTECTED>

/* PACKAGES */
PackageDeclaration            = <PACKAGE> Name
PackageDefinition             = PackageDeclaration PackageBody
PackageDefinitionOrStub       = PackageDeclaration
                                ( <SEMICOLON> | PackageBody )
PackageBody                   = <LBRACE> ( PackagedElement )* <RBRACE>
PackagedElement               = ( <DOCUMENTATION_COMMENT> )?
                                StereotypeAnnotations
                                ImportVisibilityIndicator
                                PackagedElementDefinition
PackagedElementDefinition     = PackageDefinitionOrStub
                              | ClassifierDefinitionOrStub

/***************
 * CLASSIFIERS *
 ***************/
ClassifierDefinition          = ClassDefinition
                              | ActiveClassDefinition
                              | DataTypeDefinition
                              | EnumerationDefinition
                              | AssociationDefinition
                              | SignalDefinition
                              | ActivityDefinition
ClassifierDefinitionOrStub    = ClassDefinitionOrStub
                              | ActiveClassDefinitionOrStub
                              | DataTypeDefinitionOrStub
                              | EnumerationDefinitionOrStub
                              | AssociationDefinitionOrStub
                              | SignalDefinitionOrStub
                              | ActivityDefinitionOrStub
ClassifierSignature           = Name ( TemplateParameters )?
                                ( SpecializationClause )?
TemplateParameters            = <LT> ClassifierTemplateParameter
                                ( <COMMA> ClassifierTemplateParameter )* <GT>
ClassifierTemplateParameter   = ( <DOCUMENTATION_COMMENT> )? Name
                                ( <SPECIALIZES> QualifiedName )?
SpecializationClause          = <SPECIALIZES> QualifiedNameList

/* CLASSES */
ClassDeclaration              = ( <ABSTRACT> )? <CLASS> ClassifierSignature
```

```
ClassDefinition                = ClassDeclaration ClassBody
ClassDefinitionOrStub          = ClassDeclaration ( <SEMICOLON> | ClassBody )
ClassBody                      = <LBRACE> ( ClassMember )* <RBRACE>
ClassMember                    = ( <DOCUMENTATION_COMMENT> )?
                                 StereotypeAnnotations
                                 ( VisibilityIndicator )?
                                 ClassMemberDefinition
ClassMemberDefinition          = ClassifierDefinitionOrStub
                                 | FeatureDefinitionOrStub

/* ACTIVE CLASSES */
ActiveClassDeclaration         = ( <ABSTRACT> )? <ACTIVE> <CLASS>
                                 ClassifierSignature
ActiveClassDefinition          = ActiveClassDeclaration ActiveClassBody
ActiveClassDefinitionOrStub    = ActiveClassDeclaration
                                 ( <SEMICOLON> | ActiveClassBody )
ActiveClassBody                = <LBRACE> ( ActiveClassMember )* <RBRACE>
                                 ( <DO> BehaviorClause )?
BehaviorClause                 = Block | Name
ActiveClassMember              = ( <DOCUMENTATION_COMMENT> )?
                                 StereotypeAnnotations
                                 ( VisibilityIndicator )?
                                 ActiveClassMemberDefinition
ActiveClassMemberDefinition    = ClassMemberDefinition
                               | ActiveFeatureDefinitionOrStub

/* DATA TYPES */
DataTypeDeclaration            = ( <ABSTRACT> )? <DATATYPE>
                                 ClassifierSignature
DataTypeDefinition             = DataTypeDeclaration StructuredBody
DataTypeDefinitionOrStub       = DataTypeDeclaration
                                 ( <SEMICOLON> | StructuredBody )
StructuredBody                 = <LBRACE> ( StructuredMember )* <RBRACE>
StructuredMember               = ( <DOCUMENTATION_COMMENT> )?
                                 StereotypeAnnotations ( <PUBLIC> )?
                                 PropertyDefinition

/* ASSOCIATIONS */
AssociationDeclaration         = ( <ABSTRACT> )? <ASSOC> ClassifierSignature
AssociationDefinition          = AssociationDeclaration StructuredBody
AssociationDefinitionOrStub    = AssociationDeclaration
                                 ( <SEMICOLON> | StructuredBody )

/* ENUMERATIONS */
EnumerationDeclaration         = <ENUM> Name ( SpecializationClause )?
EnumerationDefinition          = EnumerationDeclaration EnumerationBody
EnumerationDefinitionOrStub    = EnumerationDeclaration
                                 ( <SEMICOLON> | EnumerationBody )
EnumerationBody                = <LBRACE> EnumerationLiteralName
                                 ( <COMMA> EnumerationLiteralName )* <RBRACE>
EnumerationLiteralName         = ( <DOCUMENTATION_COMMENT> )? Name

/* SIGNALS */
SignalDeclaration              = ( <ABSTRACT> )? <SIGNAL> ClassifierSignature
SignalDefinition               = SignalDeclaration StructuredBody
SignalDefinitionOrStub         = SignalDeclaration
                                 ( <SEMICOLON> | StructuredBody )
```

```
/* ACTIVITIES */
ActivityDeclaration           = <ACTIVITY> Name ( TemplateParameters )?
                                FormalParameters ( <COLON> TypePart )?
ActivityDefinition            = ActivityDeclaration Block
ActivityDefinitionOrStub      = ActivityDeclaration ( <SEMICOLON> | Block )
FormalParameters              = <LPAREN> ( FormalParameterList )? <RPAREN>
FormalParameterList           = FormalParameter ( <COMMA> FormalParameter )*
FormalParameter               = ( <DOCUMENTATION_COMMENT> )?
                                StereotypeAnnotations ParameterDirection Name
                                <COLON> TypePart
ParameterDirection            = <IN> | <OUT> | <INOUT>

/* FEATURES */
FeatureDefinitionOrStub       = AttributeDefinition
                              | OperationDefinitionOrStub
ActiveFeatureDefinitionOrStub = ReceptionDefinition
                              | SignalReceptionDefinitionOrStub

/* PROPERTIES */
PropertyDefinition            = PropertyDeclaration <SEMICOLON>
AttributeDefinition           = PropertyDeclaration ( AttributeInitializer )?
                                <SEMICOLON>
AttributeInitializer          = <ASSIGN> InitializationExpression
PropertyDeclaration           = Name <COLON> ( <COMPOSE> )? TypePart
TypePart                      = TypeName ( Multiplicity )?
TypeName                      = ( QualifiedName | <ANY> )
Multiplicity                  = <LBRACKET> ( MultiplicityRange )? <RBRACKET>
                                ( <ORDERED> ( <NONUNIQUE> )?
                                | <NONUNIQUE> ( <ORDERED> )?
                                | <SEQUENCE>
                                )?
MultiplicityRange             = ( <DECIMAL_LITERAL> <DOUBLE_DOT> )?
                                UnlimitedNaturalLiteral
UnlimitedNaturalLiteral       = <DECIMAL_LITERAL> | <STAR>

/* OPERATIONS */
OperationDeclaration          = ( <ABSTRACT> )? Name FormalParameters
                                ( <COLON> TypePart )? ( RedefinitionClause )?
OperationDefinitionOrStub     = OperationDeclaration ( <SEMICOLON> | Block )
RedefinitionClause            = <REDEFINES> QualifiedNameList

/* RECEPTIONS */
ReceptionDefinition           = <RECEIVE> QualifiedName <SEMICOLON>
SignalReceptionDeclaration    = <RECEIVE> <SIGNAL> Name
                                ( SpecializationClause )?
SignalReceptionDefinitionOrStub
                              = SignalReceptionDeclaration
                                ( <SEMICOLON> | StructuredBody )
```