Human-Usable Textual Notation (HUTN) Specification

This OMG document replaces the draft adopted specification (ptc/02-10-08). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by January 15, 2003.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on April 15, 2003.

OMG Adopted Specification

Human-Usable Textual Notation (HUTN) Specification

December 2002
Final Adopted Specification
ptc/02-12-01

An Adopted Specification of the Object Management Group, Inc.

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, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ 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.

ISSUE REPORTING

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 & Specifications, Report a Bug/Issue.

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Data Access Technologies
- DSTC
- France Telecom
- IBM
- IONA
- Open_IT

- SINTEF
- Unisys

Note – The submitters wish to acknowledge the contributions of Jim Steel, Kerry Raymond and Keith Duddy of DSTC and Mariano Belaunde of France Telecom in the preparation of this specification.

Overview 1

1.1 Introduction

An HUTN standard represents an important element of the realization of the Model-Driven Architecture (MDA). This HUTN specification offers three main benefits:

- *Generic*: It is a generic specification, that can provide a concrete HUTN language for any MOF model.
- *Fully automated*: The HUTN languages can be fully automated for both production and parsing.
- *Human-Usable*: The HUTN languages are designed to conform to human-usability criteria.

1.2 Changes to Existing OMG Specifications

The HUTN Language Configuration for the expression of ECA model instances shall become part of the normative specification ptc/2002-02-05, currently entitled “UML Profile for EDOC”. See Chapter 3, “Conformance”, Section 3.4, “HutnConfig HUTN Language Configuration Conformance,” on page 3-2 for the accompanying conformance statement, which shall become a conformance point of that specification.

1.3 Proof of Concept

DSTC Pty Ltd is currently engaged in a 7 year research program into Enterprise Distributed Systems Technology with major projects devoted to enterprise modeling and the mapping of such models into middleware technology. DSTC Pty Ltd has extensive experience in the standardization, implementation and use of MOF and XMI. The DSTC has had a prototype implementation since 1999 based on XSLT [XSLT99] and Antlr [Antlr] (earlier versions used [JavaCC]) that has been used internally in

developing prototypes for other DSTC RFP Responses, and for enterprise-oriented research projects. The tool is also available for download and evaluation from http://www.dstc.edu.au/Research/Projects/Pegamento/TokTok/index.html

The original design of the system is illustrated in Figure 1-1 with the components to be implemented as part of the HUTN system shaded

The MOF Model Repository is a repository for information models, which are created in a custom model definition language (called the Meta-Object Definition Language or MODL). DSTC's MOF product is used for this purpose, since it has the advantage of being able to generate fully functional instance repositories from the model in the Model repository. The XMI subsystem performs the role of generating programs (and a DTD) for transferring data between the instance repository and XMI form.

Although the original design was quite symmetric and elegant, it was found to be easier in practice to parse directly into the instance repository rather than into XMI format. However, the original design is technically feasible.

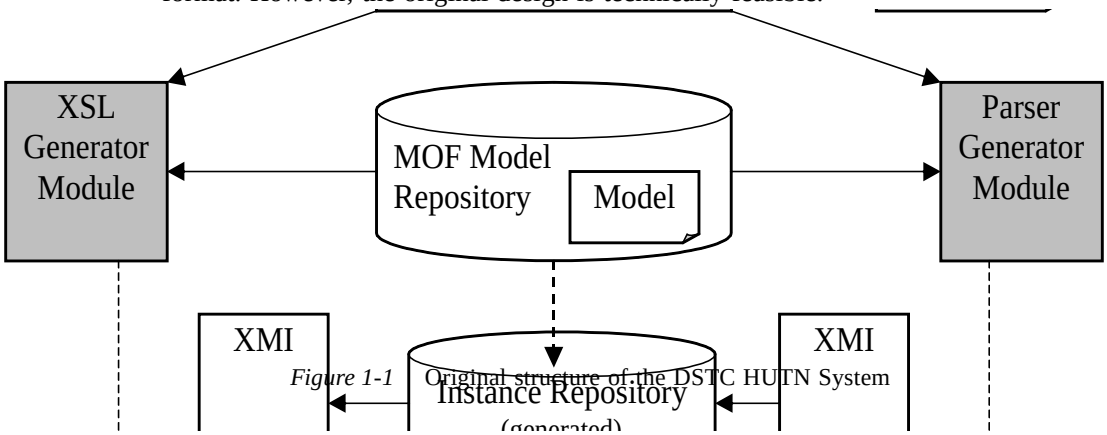

Figure 1-1 Original structure of the DSTC HUTN System

The system is divided into three basic components. The XSL generator component is responsible for the creation of an XSLT style sheet for converting a stream of XMI into the target human-usable language. The Grammar Generator component generates an ANTLR grammar and associated backend code for the parsing of the language back into a MOF-compliant repository. Finally, the so-called Configurator component is responsible for parsing a file containing the language configurations for the shorthands, and for communicating these preferences to the two generator components.

The Grammar Generator and the XSL Generator components are designed around a common generator architecture, which provides a simple mechanism for communicating with the MOF. The architecture is enacted through the use of an existing Java package included as part of the DSTC's MOF system.

The HUTN modules were implemented in the Java programming language. Java provides a number of features that make it a useful language for this purpose, such as its mature object orientation and use of interfaces, and its ready connectivity with CORBA. The CORBA product used for this implementation was Inprise's "Visibroker for Java" [Visibroker] product. This was chosen because it is the system used in the dMOF product, and was thus less likely to induce compatibility problems.

France Telecom has developed since 1997 a MOF-based model repository tool [Belaunde99] and has implemented since 1999 facilities to import and export textual human usable specifications that use a Java-like syntax (the notation was originally named JMI) and a hierarchical identification system. The implementation uses a generic parser that is connected at run-time to the API's generated from the MOF-compliant metamodel definitions (No intermediate BNF parser generation is used). France Telecom has provided feedback to the other submitters based on its original implementation.

Overall Design Rationale 2

2.1 Overall Approach

Taking the goals of the RFP and MDA into account, this specification provides a generic solution, based on generating a textual language from a MOF model. This is the same approach taken by XMI, which generates a DTD or Schema from a MOF model. This relationship between the MOF, XMI, and HUTN specifications can be seen in Figure 2-1. While XMI represents a generic serialization format for models and metamodels, HUTN is intended to be easier for human users to read and write.

The benefits of generating a HUTN language from a MOF model are:

- consistency - each HUTN generated language is different, yet they all conform to a single structure and style.
- automatable - not only can the HUTN language be generated, but also the production and parsing of text strings to/from a MOF and to/from XMI can be automated.
- completeness - anything that can be modeled in MOF (which includes all of UML) can have a HUTN language.

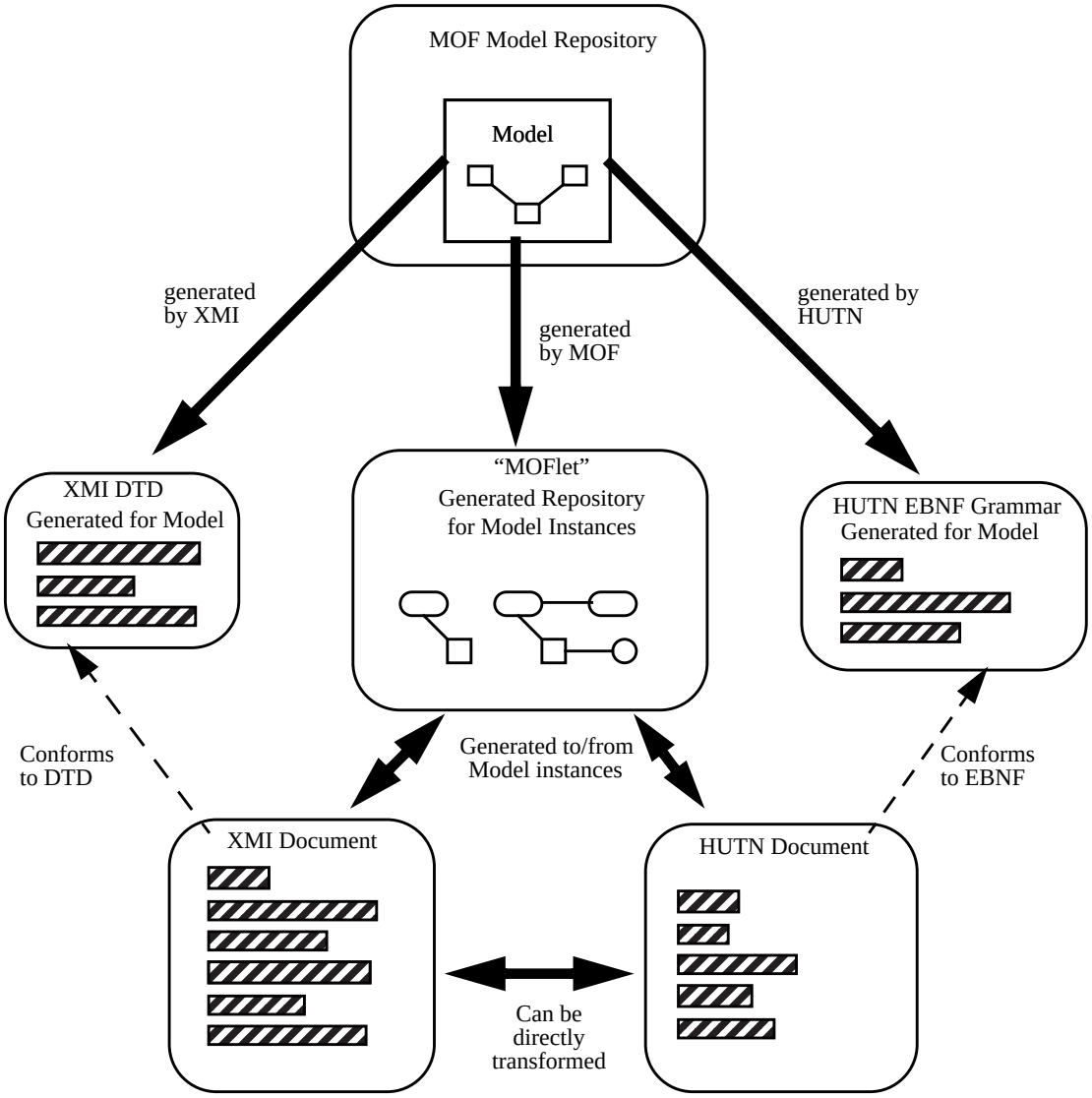

Figure 2-1 Relationship between MOF, XMI, and HUTN

2.2 Usability Criteria

The primary design goal of HUTN is human usability, and this is achieved through consideration of the successes and failures of common programming languages. HUTN uses an abstract base syntax that is applied to all models, which is customized to exploit specific properties of particular models.

As the first step in this user-centered design process, a number of assumptions had to be made about the target user audience of the generated languages. It was decided that this audience could be assumed to have some degree of familiarity with computer languages generally, while not necessarily being proficient in the use of programming languages. The syntax of a language can have a strong effect on the speed and efficiency of its use for an expert user, but the syntax features associated with this speed and efficiency often lead to a more difficult learning curve for the novice user. While it is not impossible to deal with both, a certain trade-off between these two features is apparent in many common programming languages. For example, the C programming language features many syntactic elements that are convenient for the experienced user, but the language is widely acknowledged as one of the more difficult to learn. By contrast, the Pascal language is a very popular language for teaching programming, but is less popular for large-scale development, where it is more time consuming and less efficient than a language such as the C programming language. For this application, it was decided that an efficient learning curve was a more important requirement of the languages, and that they would consequently be designed with learnability as a primary goal, and expert-friendliness as a secondary goal.

2.2.1 Syntax and Aesthetics

There is a proliferation of opinions on the aesthetic virtues and downfalls of programming languages, and of what features are important when designing a language. However, while there is an abundance of papers on the design of a language's semantic operations, there are surprisingly few published works on programming language usability as it pertains purely to syntax. It must also be considered that there are essential differences between the syntactic structure and features of a programming language and those that the HUTN languages might contain. While a programming language is aimed at the modification and maintenance of a (usually abstract) body of information, the HUTN languages are required purely for the display of information. For this reason, programming language features such as control constructs have no real relevance to HUTN languages.

Two usability works on programming languages and their syntax were considered to assist in identifying principles upon which to design the HUTN languages. The first [McIver96] is a paper by McIver and Conway from Monash University, which identifies and explore problems associated with languages used as first languages for the teaching of programming. The paper also discusses a number of directions for the design of such a language. The other [RL77] is a paper by Richard and Ledgard of the University of Massachusetts, and discussed a number of principles for syntax design, with a view to designing a general purpose programming language called Utopia84. From these papers and from independent consideration, a number of principles have been assembled for the design of the HUTN language.

2.2.2 Use of symbols and punctuation

An important principle of human-usability is that a language should have sufficient variety of symbols that the user should be able to easily navigate through a stream of data. A language that does not follow this principle is the LISP language which uses parentheses almost exclusively. This problem often comes about from a language's

devotion to a certain functional, logical, or object-oriented paradigm. As pointed out by McIver and Conway, this can also lead to the problem of ‘syntactic homonyms’, the use of a single syntactic construct to represent two distinct semantics. Richard and Ledgard also identify this as a problem, emphasizing that “distinct features should have distinct forms.”

However, the reverse of this can also be a problem. A language that makes use of a large vocabulary can make a novice’s task of learning the language very difficult, and often misleading. The Ada 9x languages, for example, have 68 reserved words and over 50 predefined attributes. As explained by McIver and Conway, this problem is often dealt with by teaching the learner only a small subset of the language’s vocabulary. However, this can lead to confusion when the student is exposed to the new features of the language, and can lead to the production of overly verbose or obscure code if they neglect to use some language features. The problem can also lead to the presence of ‘syntactic synonyms’, the availability of a number of syntactic constructs for the presentation of a single construct. These synonyms only serve to further mislead the student and unnecessarily expand the vocabulary of a language.

Related to this problem is the excessive use of symbols for the denotation of functions or, to a lesser degree, for the denotation of syntactic structure. This is evident in languages such as C, particularly. While the resultant terse syntax can make the language very efficient for expert users, it has a detrimental effect on the novice user’s ability to learn the language.

2.2.3 Use of reserved words

Another language syndrome to be avoided is the overuse of natural language words for syntactic structuring. While not as significant a problem as terse syntax, the verbose syntax that can result in a language that becomes harder to read by virtue of the sheer bulk of information being presented. Also, symbols are more intuitive delimiters of structure than words, since natural languages use symbols exclusively for punctuation. This division between words for semantic functions and symbols for punctuation is a useful general rule, in part because of the ties with natural language, and in part because of the roles that words and symbols play. Words are useful when their function requires a degree of explanation, whereas structure delimitation requires little such explanation, so is better suited to a more brief representation.

2.2.4 User expectations

One of the programming language faults identified by McIver and Conway is that of backward compatibility. They define backward compatibility in two forms: genetic compatibility and mimetic compatibility. Genetic compatibility refers to syntactic similarities in programming languages that result from one language being developed as a successor to the first (such as C and C++). Mimetic compatibility, by contrast, refers to language features that are derived from de-facto standards, such as the use of square brackets for indexing into arrays. The authors suggest that both of these were too often agents for the propagation of syntactic features that, while familiar to those with programming experience, conflicted with a novice’s preconceived ideas of what a function might appear as. However, since the target audience of a HUTN generated-

language is assumed to have some familiarity with programming conventions, the situation is reversed. These syntactic familiarities can serve the purpose of providing the new user with a head start in learning the language.

The final usability consideration taken from the two papers was avoiding the problem of violating the user's expectations. This often comes about through poor selection of function names and appearances. In some situations, the orthogonality of concepts can mean that misleading code can arise through the obscure and complicated combination of simple features. However, the names used in a HUTN-generated language come from the underlying model, which presumably conforms to the user's expectations.

2.2.5 Other considerations

Indentation plays an important part in improving the readability of textual documents, and particularly in enhancing the navigability of programming language source code. This is also the case in the HUTN languages, and an indentation policy should be incorporated into the producer of HUTN text.

It is quite probable that the users of the HUTN languages will be involved in the use of a number of HUTN languages, either through the evolution of a single domain model or through the use of a number of models. This implies a need for some uniformity between the HUTN languages. This is achieved through the use of a common basic structure for the languages, the design of which is described in Section 4.1, "Overview," on page 4-1.

One of the major decisions made to enhance the usability of the HUTN-generated languages was to allow the use of alternative forms (or short-hands). These configurations would involve simple syntactic extensions without changing either the larger syntactic structure or the semantics of the language. The details of these short-hands are described in Section 4.2, "The Base Language," on page 4-1.

2.3 The Meta-Object Facility (MOF)

OMG's Meta-Object Facility (MOF, formal/00-04-03) specifies a small but complete set of modeling concepts that can be used to express information models. In line with the OMG's commitment to CORBA, the MOF standard also provides a mapping from these modeling concepts to CORBA IDL to support a repository of instances of that model. Although not part of the MOF standard, some MOF tools (such as DSTC's dMOF product) also generate the code for the model-instance repository.

There are a number of essential concepts used in MOF modeling. A *Package* is used to encapsulate a collection of related Classes and Associations. Packages can also contain simple type definitions, equivalent to those available in CORBA IDL. *Classes* exist in the commonly-used sense of the word, describing an object and its properties. These properties are represented through Attributes and References, which can be inherited using a multiple-inheritance system based on that of CORBA IDL. *Attributes* have a name and a type, selected from the CORBA type system[1]. This includes a range of types from basic types such as integers, strings and booleans, to more complex types such as enumerations, and through to structured types. In addition, attributes have both upper and lower limits on the number of times that they can appear within a class

instance. An *Association* is used to represent a relationship between instances of two classes, each of which plays a *role* within the association. Associations can have the additional property of *containment*; an association represents a *containment relationship* if one of the participant classes does not exist outside the scope of the other. A Class participating in an association can also contain a *Reference* to the association. A reference appears much like an attribute, but reflects the set of class instances that participate in the Association with the containing class instance.

2.4 XML-based Model Interchange (XMI)

The XML-based Model Interchange (XMI) Format standard [XMI98] defines a set of mappings from the MOF modeling concepts to a representation in XML (eXtensible Markup Language), a standard of the World Wide Web Consortium (W3C) [XML98].

XML was chosen for its growing popularity for data expression, and for the flexibility provided by its type definition system. The XML is essentially a tree-based language consisting of a series of nested “elements,” each of which is represented by a set of matching start and end tags. These elements may also include a number of name-value pairs called attributes, which appear within the opening tag of the element. The flexibility of the language lies in the ability to associate an XML document with a Document Type Definition (DTD). This DTD allows for the placement of further specific restrictions on the contents of an element. These include restrictions on the type of data (for example, numbers, strings with/without white space) allowable between two tags. The element can also be restricted in terms of the attributes that may appear within the element, and on the types of their value. Further, a restriction can be placed on the different elements (and the number of each) that are allowable beneath an element on the document tree.

The XMI specification provides two main components: a set of rules for producing a DTD from a model, and a set of rules for the transfer of data between XMI and a MOF-compliant repository. Each instance of a MOF Package, Class, or Association is represented by an XML element. In addition, every instance of a MOF Class contains an XMI identifier in the form of an attribute labelled “xmi.id” on the instance’s XML element. When a class instance appears by reference (rather in the form of a full declaration), it is referenced by an “xmi.idref” attribute in the XML element. MOF Attributes whose types are simple types are represented as elements containing data, except for enumerations and booleans, whose values are enclosed in attributes, within self-closing tags. Attributes whose values are class instances are represented either as class instance declarations or as references to class instances using the scheme mentioned above.

---

1. The type system for MOF Attributes is currently the subject of revision within the MOF RTF and may change during the lifetime of this RFP.

2.5 Example MOF Model

To illustrate the MOF, XMI, HUTN relationship with a concrete example, consider the MOF model in Figure 2-2 describing a family. Section 2.6 gives an example of an XMI stream for that model, describing a number of families. Section 2.7 represents the same information in the HUTN-generated language for this model. As can easily be seen, the HUTN is much more human-readable than the XMI.

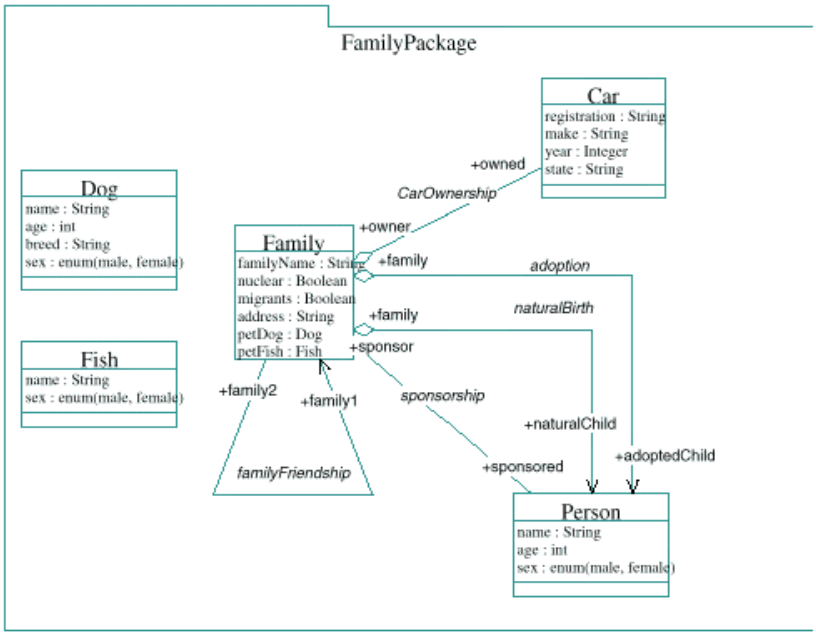

Figure 2-2 MOF model for a family

2.6 Example XMI

The following XML stream represents an instance of the model in Section 2.5.

```
<?xml version = "1.0"?>
<XMI>
 <XMI.header>
  <XMI.model xmi.name = ‘familyPackage’ xmi.version = ‘1.1’/>
 </XMI.header>
 <XMI.content>
  <FamilyPackage xmi.id=’xmi-id-001’>
   <FamilyPackage.Family xmi.id=‘xmi-id-002’>
    <FamilyPackage.Family.familyName>
     The McDonalds
    </FamilyPackage.Family.familyName>
    <FamilyPackage.Family.address>
```

```
    7 Main Street
  </FamilyPackage.Family.address>
  <FamilyPackage.Family.nuclear xmi.value='false'/>
  <FamilyPackage.Family.migrants xmi.value='true'/>
  <FamilyPackage.Family.familyFriends>
    <FamilyPackage.Family xmi.idref='xmi-id-003'/>
  </FamilyPackage.Family.familyFriends>
  <FamilyPackage.Family.petFish>
    <FamilyPackage.Fish>
      <FamilyPackage.Fish.name>
        Wanda
      </FamilyPackage.Fish.name>
      <FamilyPackage.Fish.sex xmi.value='female'/>
    </FamilyPackage.Fish>
  </Familypackage.Family.petfish>
  <FamilyPackage.Family.petDog>
    <FamilyPackage.Family.Dog xmi.idref='xmi-id-007'/>
  </FamilyPackage.Family.petDog>
</FamilyPackage.Family>
<FamilyPackage.Family xmi.id='xmi-id-003'>
  <FamilyPackage.Family.nuclear xmi.value='true'/>
  <FamilyPackage.Family.migrants xmi.value='false'/>
  <FamilyPackage.Family.address>
    5 Main Street, Brisbane
  </FamilyPackage.Family.address>
  <FamilyPackage.Family.familyName>
    The Smiths
  </FamilyPackage.Family.familyName>
  <FamilyPackage.Family.naturalChild>
    <FamilyPackage.Person>
      <FamilyPackage.Person.name>
        Joan Smith
      </FamilyPackage.Person.name>
      <FamilyPackage.Person.age>
        20
      </FamilyPackage.Person.age>
      <FamilyPackage.Person.sex xmi.value='female'/>
    </FamilyPackage.Person>
  </FamilyPackage.Family.naturalChild>
  <FamilyPackage.Family.naturalChild>
    <FamilyPackage.Person>
      <FamilyPackage.Person.name>
        Harry Smith
      </FamilyPackage.Person.name>
      <FamilyPackage.Person.age>
        17
      </FamilyPackage.Person.age>
      <FamilyPackage.Person.sex xmi.value='male'/>
    </FamilyPackage.Person>
  </FamilyPackage.Family.naturalChild>
  <FamilyPackage.Family.adoptedChild>
```

```
    <FamilyPackage.Person>
      <FamilyPackage.Person.name>
        Dylan Smith
      </FamilyPackage.Person.name>
      <FamilyPackage.Person.age>
        12
      </FamilyPackage.Person.age>
      <FamilyPackage.Person.sex xmi.value='male'/>
    </FamilyPackage.Person>
  </FamilyPackage.Family.adoptedChild>
  <FamilyPackage.Family.familyFriends>
    <FamilyPackage.Family xmi.idref='xmi-id-002'/>
  </FamilyPackage.Family.familyFriends>
</FamilyPackage.Family>
<FamilyPackage.Person xmi.id='xmi-id-004'>
  <FamilyPackage.Person.sex xmi.value='male'/>
  <FamilyPackage.Person.age>
    7
  </FamilyPackage.Person.age>
  <FamilyPackage.Person.name>
    Namdou Ndiaye
  </FamilyPackage.Person.name>
</FamilyPackage.Person>
<FamilyPackage.Person xmi.id='xmi-id-005'>
  <FamilyPackage.Person.sex xmi.value='male'/>
  <FamilyPackage.Person.age>
    6
  </FamilyPackage.Person.age>
  <FamilyPackage.Person.name>
    Sharif Mbangwa
  </FamilyPackage.Person.name>
</FamilyPackage.Person>
<FamilyPackage.Person xmi.id='xmi-id-006'>
  <FamilyPackage.Person.sex xmi.value='male'/>
  <FamilyPackage.Person.age>
    3
  </FamilyPackage.Person.age>
  <FamilyPackage.Person.name>
    Miguel Aranjuez
  </FamilyPackage.Person.name>
</FamilyPackage.Person>
<FamilyPackage.Dog xmi.id='xmi-id-007'>
  <FamilyPackage.Dog.sex xmi.value='male'/>
  <FamilyPackage.Dog.age>
    2
  </FamilyPackage.Dog.age>
  <FamilyPackage.Dog.name>
    Spike
  </FamilyPackage.Dog.name>
  <FamilyPackage.Dog.breed>
    Irish Wolfhound
```

```
    </FamilyPackage.Dog.breed>
  </FamilyPackage.Dog>
  <FamilyPackage.Sponsorship>
    <FamilyPackage.Family xmi.idref='xmi-id-003'/>
    <FamilyPackage.Person xmi.idref='xmi-id-004'/>
    <FamilyPackage.Family xmi.idref='xmi-id-003'/>
    <FamilyPackage.Person xmi.idref='xmi-id-005'/>
    <FamilyPackage.Family xmi.idref='xmi-id-002'/>
    <FamilyPackage.Person xmi.idref='xmi-id-006'/>
  </FamilyPackage.Sponsorship>
  <FamilyPackage.CarOwnership>
    <FamilyPackage.Family xmi.idref='xmi-id-002'/>
    <FamilyPackage.Car>
      <FamilyPackage.Car.Registration>
        755-BDL
      </FamilyPackage.Car.Registration>
      <FamilyPackage.Car.State>
        QLD
      </FamilyPackage.Car.State>
      <FamilyPackage.Car.Make>
        Mitsubishi Magna
      </FamilyPackage.Car.Make>
      <FamilyPackage.Car.Year>
        1992
      </FamilyPackage.Car.Year>
    </FamilyPackage.Car>
  </FamilyPackage.CarOwnership>
 </FamilyPackage>
</XMI.content>
</XMI>
```

Figure 2-3 An example XMI stream for two families

As Figure 2-3 clearly demonstrates, the XMI/XML format is one that is neither succinct, nor easily readable or writable. Although the XMI standard is still under revision, the basic structure of the language and its ties with XML will not change and, as such, these human usability problems are likely to remain.

2.7 Equivalent HUTN

The following text is the HUTN-generated equivalent representation of the same example as in Section 2.6.

```
FamilyPackage id-001 {

    Family "The McDonalds" {
        address: "7 Main Street"
        migrants
        familyFriends: "The Smiths"
        petFish: female Fish "Wanda";
```

```
        petDog: “Spike”
        CarOwnership: “755-BDL” {
            state: QLD
            make: “Mitsubishi Magna”
            year: 1992
        }
    }

    nuclear Family “The Smiths” {
        address: “5 Main Street”
        naturalChild: female Person “Joan Smith” {
            age: 20
        }
        naturalChild: male Person “Harry Smith” {
            age: 17
        }
        adoptedChild: male Person “Dylan Smith” {
            age: 12
        }
        familyFriends: “The McDonalds”
    }

    male Person “Namdou Ndiaye” {
        age: 6
    }

    male Person “Sharif Mbangwa” {
        age: 3
    }

    male Person “Miguel Aranjuez” {
        age: 2
    }

    male Dog “Spike” {
        age: 2
        breed: “Irish Wolfhound”
    }

    sponsorship {
        “The Smiths”            “Namdou Ndiaye”
        “The Smiths”            “Sharif Mbangwa”
        “The McDonalds”         “Miguel Aranjuez”
    }
}
```

Figure 2-4 The same example in the HUTN-generated language

2.8 Summary

The stark contrast between the example XMI/XML in Section 2.6 and the equivalent HUTN in Section 2.7 shows that a language generation facility designed with sufficient consideration of usability can make significant advances in providing a human-usable mechanism for the interchange of data with repositories. This usability comes about not by coincidence, but through the adoption of a user-centric design approach, considering the needs of the user before the technical agenda of the system's development. The alignment of the generated language's style with those of common programming languages provides the user with a familiar frame of reference for learning the language. Also, careful consideration of the problems associated with existing programming languages' styles leads to a syntax that will be able to avoid these problems.

There are three properties that make the generation of HUTN languages a particularly useful. The first is that it is generic, in that it can provide a language for any model that can be specified using the MOF techniques. Secondly, the HUTN specification can be fully automated, particularly useful for systems whose information models are undergoing rapid change. Thirdly, the family of HUTN languages were designed to conform to human-usability criteria.

2.8.1 Generic

The language mappings as described in Chapter 6 provide a set of syntactic rules providing complete coverage for all of the MOF modeling concepts. This means that a language can be rapidly created for any model specified using these concepts. In addition, since the MOF modeling concepts have been designed as a basic set of common concepts, there will almost always be a simple mapping from these concepts to alternative modeling techniques. Therefore it should be possible to use the syntax described to develop a similar system for other modeling and repository tools.

Since the HUTN mappings are based on transformations to (and potentially from) MOF, it can also be used to translate to/from XMI.

2.8.2 Fully Automated

The second benefit of the generation of the HUTN languages is that it can be fully automated (see Section 1.3, "Proof of Concept," on page 1-1 for a description of DSTC's prototype). The task involved in the manual implementation of a parser allows for more flexibility in language design, but requires a good deal of time and effort. In addition to this, a manually constructed parser is open to problems with information models that are subject to change. Automation means that changes made to a language, be they as a result of a change in the underlying model or a change in the syntax, will be implemented uniformly and quickly across the entire system. In this way, automation avoids problems of consistency in changing languages, and greatly reduces the time involved in the evolution of an information model/repository suite.

2.8.3 Human Usable

The other major benefit provided by the HUTN languages is the human-usability. While the essential style of the language is fixed and hence familiar, the individual generated languages are specific to each model and incorporate model-specific shorthands.

Conformance 3

3.1 Overview

There are three conformance points for Chapter 4, “HUTN Design Rationale” up to and including Chapter 7, “Configuration Notation” that apply to HUTN tool implementations:

1. Input text stream conformance (defined in section 3.2).
2. Output text stream conformance (defined in section 3.3).
3. HUTN Configuration Documents (defined in section 3.4).

In addition, there are further compliance points that will relate to the UML Profile for EDOC, as amended by Chapter 8 (defined in section 3.5).

3.2 Input Stream Conformance

For all given combinations of MOF models and HUTN Configurations, a HUTN parser must be able to recognize any legal HUTN document, as defined in this specification.

Note – This implies that a conformant HUTN parser for a given combination of MOF model and HUTN Configuration must recognize the input from all conformant HUTN document generators for the some model/configuration pair.

3.3 Output Stream Conformance

For a given combination of MOF model and HUTN Configuration, a HUTN document generator must be able to output at least one legal form of HUTN document, as defined in this specification.

Note – The capacity to generate all alternative forms and the internal heuristics or external instructions or influences used to determine the choice between them is an area for product differentiation.

3.4 HutnConfig HUTN Language Configuration Conformance

The HUTN language configuration for the org.omg.HutnConfig MOF model given in Chapter 7 is normative for input to HUTN tools which parse HUTN Configurations. It is also the standard representation for “@Config” comments in HUTN documents, and shall be acceptable to tools which parse these comments.

3.5 ECA HUTN Language Configuration Conformance

The HUTN language configuration for the org.omg.ECA MOF model given in Chapter 8 is normative for HUTN tools which parse ECA model instances expressed as HUTN documents, or output model instances as HUTN documents.

HUTN Design Rationale 4

4.1 Overview

The generation of HUTN languages starts with an abstract base language, which is then customized by the use of model-specific information.

4.2 The Base Language

4.2.1 Use of familiar forms

Using the structural and syntactic features of existing languages is a good way to enhance the learnability of the HUTN languages, and to ensure that the user's expectations are not violated. To this end, HUTN languages use syntactic forms drawn from CORBA IDL and structural forms taken from XMI (XML), as HUTN users are believed to be familiar with these.

4.2.2 Structure reflects containment

Languages, on the whole, represent information in a fairly similar way. A document invariably consists of a set of concepts, each of which consists of a number of other concepts, and so on until the concepts are nothing but simple pieces of atomic data. This can be seen in both procedural and object-oriented programming languages, as well as in natural English. For example, an English essay could be said to consist of a series of paragraphs, each of which contains a series of sentences, which contain a series of words. A piece of source code for the Java programming language [Java] could consist of a series of import statements, package statements, and class definitions, which contain variables and methods, which contain sets of parameters and statements.

At different levels of depth on this ‘concept tree,’ the representation of the containing concept changes. One common change is for concepts higher on this tree to be introduced in some way. For example the essay with its paragraphs might first have a title, or chapters within a thesis might have chapter numbers and titles. A method declaration in a Java class definition has a visibility value, a method name, and a return type. By contrast, where an element is the only possible element in its position, it may go without an introduction, such as sentences within a paragraph, or statements within a Java method definition. However, to be effective this requires some language familiarity on the part of the user, something that cannot be assumed for the HUTN languages.

Particularly in structured notations such as programming languages, it is often necessary to separate the contained concepts using some form of punctuation. Java, for example, uses braces to delimit method bodies, commas to separate method parameters, and semicolons to terminate statements. Written English uses full stops to terminate sentences, and commas or parentheses to delimit phrases. The choice of symbols for separating punctuation can also be dependent on the depth of the concept on the tree. For example, braces are often associated in programming languages with high-level or major concepts such as procedure declarations, while commas are often associated with low-level or minor ones, such as a list of method parameters.

The MOF modeling concepts underlying the HUTN languages also conform to this ‘concept tree’ paradigm. Package instances contain Class instances, Class instances contain Attribute values, and so on. Accordingly, the HUTN language core has been based around these ideas of concept containment, introduction and delimitation.

4.2.3 Defining and referencing major concepts

The MOF Package, Class, and Association concepts have been classified as ‘major’ concepts, warranting an introduction for their instances. The introduction is a simple one, consisting of the name of the Class, Package, or Association and some identifying string. (When translating from XMI, the XMI ID provides a logical and automatically unique identifier). The appearance of this introduction is very similar to the introductions of procedures or functions in Pascal or C. Curly braces, as used in many languages deriving syntactic features from C, are used to delimit the bodies of these major concepts. Class instances can also be referenced by other parts of the document. This is done by simply displaying the introduction of the instance without the body.

4.2.4 Representing minor concepts

By contrast, MOF Attributes are denoted as minor concepts, and as such are represented differently. In their case, the attribute name is followed by a colon or equal sign, followed in turn by the value of the attribute. The attributes’ representations may be separated only by white space, or with a semi-colon terminator. White-space-only separation is possible because it is always feasible to know how many white-space separated ‘words’ will appear in an attribute’s value. No simple attributes are permitted white space within their values except string-typed attributes, whose values are delimited by a number of possible delimiting characters, or left undelimited, if their contents make this possible. Attributes whose values are class instances are represented

either as instance references or as full instance declarations, depending on the nature of the attribute. These representations do have more than one ‘word’ in their value, but do not cause problems because the number of words is always fixed and known to the parser.

References are displayed with the reference name followed by a colon or equal sign and the representation of the class instance that is referred to. This is almost identical to the representation of attributes, which could be seen as violating the principle of ‘different forms for different features.’ However, the role of references in the MOF is in many ways to provide a class instance with attribute-like access to other class-instances that are related by association links. For this reason, the underlying ‘feature’ of references and class-instance valued attributes is essentially the same, and thus their representations should in fact be similar.

4.3 Model-Specific Shorthands

There are several kinds of model-specific shorthands and configurations available in HUTN, each described in the following subsections:

- The use of a class’s attribute as the class’s unique identifier, and the specification of the scope over which the identifier is unique.
- The representation of a boolean or enumerated attribute as a keyword or adjective in the Class header rather than a name-value pair in its body.
- The omission of the class type of an object reference when only one type is possible, or the omission of the reference name for containment relationships.
- The use of default values for mandatory attributes, enabling them to be omitted in many cases.
- Alternative representations for associations.
- The selection of an alternate name for any model element for HUTN language-generation purposes.
- The use of *parametric form* for attribute values; that is, representing a number of a class’s attribute values in parenthesis in the Class header rather than as name-value pairs in the body.

Some of these short-hands can be incorporated automatically from analysis of the MOF model, but a couple of them require some additional information about the model. Those that do are specified using a language configuration MOF-metamodel. This metamodel is presented in the next section, and is followed by descriptions of the various available shorthands. The final section will discuss the effects of inheritance on the various available configuration.

4.3.1 Identifying class instances (objects)

Class instances (objects) are concepts that can be referred to by other constructs, such as References, Associations, and Attributes. For this reason they require a unique name by which they can be identified within the HUTN text stream. As mentioned above, an

arbitrary unique identifier such as the XMI ID provides this, and is thus a logical choice for a default identifier. However, since this string is meaningless in terms of the data being represented, it makes for a somewhat poor identifier with regard to usability. It would be far better to use an identifier that somehow is relevant to or symbolic of the instance that it identifies.

The logical choice, therefore, is to use an attribute of the class instance as the identifier. Many classes do contain an attribute which can serve as a unique identifier for the object (e.g., a name or ID field). However, it is vital that the attribute chosen does indeed distinguish between the instances of this class (it does not have to be distinct relative to instances of other unrelated classes). Since the MOF does not provide a mechanism by which an attribute can be defined as unique, the responsibility of ensuring the uniqueness of the identifying attribute must be provided by the user/model-builder.

For attributes with a very limited range of values, this is obviously difficult. For example, if an attribute's type is boolean, or if it is an enumeration, the attribute is unlikely to make a useful identifier, since its small range of values will greatly restrict the number of instances that it can uniquely identify. However, there are cases where these types may serve to uniquely identify class instances. The only attributes that are never practical as unique identifiers, due mainly to the impossibility of comparing values, are those of class types, and therefore it is illegal to nominate a class-valued attribute as a unique identifier for a Class. Also, the attribute selected must be mandatory (not optional), since it will always be required to generate the object identifier.

This identifying attribute may not be configured to have a default value, as only a single instance of a class (within a given scope) may have any particular value for this attribute if it is an identifying attribute. (See section 4.3.5 for information about configuration of default values.)

Since the value of the attribute selected as the identifying attribute is presented in the class instance's introduction, its normal representation within the body of the instance's definition is superfluous, and is thus omitted.

In addition to defining the meaning of the identifying string, users are also given the ability to define the scope over which the string is unique. By default, identifying strings (be they arbitrary or the value of an attribute) are required to be unique over the set of instances of the class, and over all instances of its subtypes. Alternatively, the identifier may be configured to be unique within the scope of the instance's containing object. This allows for nested identification structures.

Class instance identifiers are configured using the IdentifierConfig metaclass (see section 5.1.6).

4.3.2 Keywords and Adjectives

When a boolean attribute is mandatory, it seems redundant to display the attribute's name as well as its value each time it is displayed. The display of two pieces of information seems unnecessary for the representation of a variable that only has two states. A shorthand is to display the attribute only if it is true, and to elide its

representation if it is false. Further to this, since the presence of the representation already denotes that the attribute's value is true, it would be far more efficient to simply the display the name of the attribute. We call the use of such a name alone in the body of a class-instance a *keyword*.

However, the use of keywords is restricted to attributes that are mandatory and do not contain more than one value. There are only two states available to a keyword representation: present or absent. By contrast, an optional boolean has three plausible states: true, false, and not defined. Attributes with more than one value obviously have even more than this, and as such neither optional nor multiply-defined boolean attributes can be shortened to keywords.

In the absence of a default value for a mandatory single-valued boolean attribute, it is assumed to be false, and its name will appear in the class instance header only if this is not the case (the value is true). This is also the case if a default value of false is explicitly configured for this attribute. (See section 4.3.5). If, however, the attribute is configured with a default value of true, then the tilde symbol '~' is used before the attribute name keyword to denote that its value is false, and the absence of the attribute name indicates a value of true.

Programming languages such as C++, Java and Pascal represent various pieces of information other than an identifier in the definition of methods or procedures. These can include the visibility of the method, or the return type of the method. Variables displayed in this way are called *adjectives*, since they provide information about an object before it is declared, much like an adjective describing a noun in the English grammar.

Adjectives in HUTN are similar to keywords, but are located differently within the representation of the class instance. Where a keyword's representation is expressed in the body, an adjective's is placed directly before the name of the class in the introduction of the declaration. There are two kinds of adjectives: boolean-valued attributes, and enumeration-valued attributes.

Like keywords, boolean-valued adjectives are restricted to single-valued mandatory boolean attributes, and the name of the attribute (perhaps preceded by a tilde symbol) is used as the adjective. Enumeration-valued adjectives, however, use the enumeration labels as the adjective. While boolean-valued adjectives are available by default, enumeration-valued adjectives must be configured. The metaclass for configuring enumeration-valued adjectives is EnumAdjectiveConfig (see section 5.1.7).

Like the identifying attribute of a class, an attribute that is represented as adjective, within the introduction of a class instance, need not be shown again within the body of the class instance.

When one or more attributes are defined as adjectives on a class, the nature of the parser may be required to change. When no adjectives are present in a language, a parser may use a look-ahead of only one symbol, since the next occurring symbol will uniquely determine the current state of the parser. However, the introduction of adjectives changes this. For example, if two classes are given attributes of the same name, or if an adjective is inherited from a class's parent class, then the presence of the adjective in the symbol stream will not be sufficient to determine which class the

forthcoming instance will belong to. For this reason, it is necessary to make the look-ahead of the parser greater than one. More exactly, the look-ahead must be greater than the number of adjectives on that class that has the most shared adjectives.

Other conflicts are possible when using enumeration-valued attributes as adjectives, such as a name clash between enumeration labels for two different attributes or a name clash between an enumeration label and the name of a boolean-valued adjective. See Section 4.3.7, “Renaming of Model Elements for HUTN languages,” on page 4-7, for ways of overcoming such conflicts.

4.3.3 Omission of Class Type of an Object Reference

The default form of referring to a class instance (object) is to give its class name and its object identifier. However, due to the strong typing of the MOF, there are many situations in which the references will always be to a known class and hence the class name can be omitted.

There are three situations when this shorthand can be used; in a MOF Reference, in an attribute where the type of the attribute is an instance of a local MOF class, and in the representation of an association. Each of these is subject to two conditions. If the object that is being referred to is contained by the referring object, then the referred object may be represented as a declaration rather than a reference and, in this case, its type name may not be omitted.

The second condition is that the referred class and all of its subclasses must use a consistent identification scheme, either by all using the same attribute as an identifying attribute, or all arbitrary unique identifiers. Without this condition, the risk would be introduced of having two objects with the same identifying string.

4.3.4 Omission of Reference Name for a Contained Object

When a contained object is defined within its container object, the default language rules require the name of the MOF Reference to identify the containment association involved. However, in practice, there is often only one containment association from the container class to the contained class, and therefore the MOF Reference name can be omitted. When there are two containment associations by which a contained object and its container may be linked, the omission of the reference name is not permitted.

4.3.5 Default Values

Often a class has an attribute for which many of the class instances will assign the same value to that attribute. Reading and write this same attribute-value pair is very tedious. Therefore, the HUTN supports the shorthand of omitting the attribute-value pair if the value is intended to be the default value. Note that the default value cannot be determined from the MOF model and so user/modeler input is needed. Default values for an attribute on a class are configured using the DefaultValueConfig metaclass (see section 5.1.8).

The effect of this on a HUTN producer is that the attribute's representation need not be shown if its value is the same as its default value. For a consumer, the absence of the attribute in the object declaration can be taken to mean that it has as its value the attribute's default value.

Since classes that use an attribute as an identifier must show be identified, an attribute selected as such may not be assigned a default value. Similarly, attributes configured for parametric display (as described in section 4.3.6) are also not configurable for a default value. As described in section 4.3.2, the use of a default value on a single-valued boolean-valued attribute affects the behavior of it keyword and adjective forms.

4.3.6 Parametric Form

Some class instances have a conventional order for the values of certain attributes, which makes it unnecessary to give the names of those attributes. For example cartesian coordinates have numeric values named X and Y, but are often represented as two comma separated numbers (2, 4), where it is well known that the first value represents X and the second represents Y. The form of their representation resembles actual parameters to a function or method, and is hence called *parametric form*.

A configuration of a class for a HUTN language may include a list of attributes whose values will appear in the class instance header, in parentheses after the class identifier, and before the class body. The parametric attribute values will be represented in the same way as multi-valued attributes; that is, separated by whitespace or commas. These attributes will then be omitted from the class body.

To make the parametric form simple and consistent, attributes nominated by a parametric configuration may not be multi-valued, and may not be optional. In addition, these attributes may not have default values, as each class instance will have explicit values given for each parametric attribute every time.

Attributes having parametric form within a class are configured using the ParametricConfig metaclass (see section 5.1.9).

4.3.7 Renaming of Model Elements for HUTN languages

In human readable textual representations of MOF Models it is sometimes useful to use simpler names for model elements than used in other representations. Some examples include the elimination of white space, and the shortening of long names. In addition it may be useful to allow certain shorthands given above by removing name clashes that may arise when placing previously separated names into the same namespace. For example, a MOF model may have the same name for an enumeration label as for a boolean attribute name. In this case it would be impossible to use both an attribute with the enumeration as a type and the boolean attribute as adjectives for the same class configuration. However, by renaming either the enumeration label or the boolean attribute name, the clash can be avoided.

A renaming configuration allows any MOF Model Element whose name has a rendering in HUTN (Package, Class, Attribute, Reference, Association, Association End, Enumeration label) to be assigned a new name for the purposes of configuring a HUTN representation of the model. This is done using an instance of the RenameConfig metaclass (see section 5.1.10).

Configuration 5

Configuration options for HUTN languages are expressed as instances of the HutnConfig metamodel, which is described in this chapter.

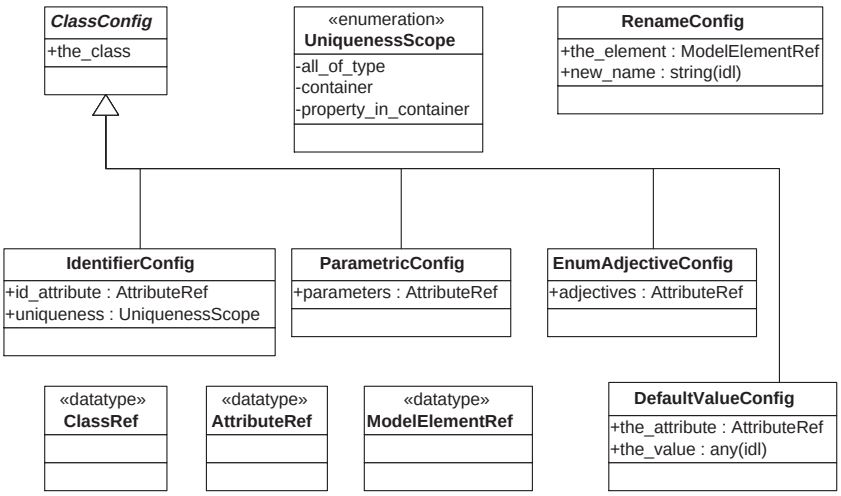

Figure 5-1 The HutnConfig MOF metamodel

5.1 HutnConfig Metamodel

This is the HutnConfig metamodel. The semantics of each element are described in the following sections. RenameConfig applies to any Model Element whose name has can be represented in a HUTN language (Class, Association, Reference, Attribute, Package, Enumeration label, Association End), while the other configurations apply to a Class.

5.1.1 ClassConfig

This metaclass is abstract, and is inherited by all the concrete configuration metaclasses which relate to the textual representation of MOF Classes. The exception to this is the RenameConfig metaclass, which may refer to any MOF Model Element.

the_class

This metaattribute is a reference to the MOF Class being configured,

5.1.2 «enumeration» UniquenessScope

This enumeration gives possible scopes for uniqueness of identifying attributes of class instances. It is used in the IdentifierConfig metaclass (see section 5.1.6).

all_of_type

This value indicates that the scope for uniqueness of attribute values identifying class instances is all instances of the class and all instances of subtypes of the class.

container

This value indicates that the scope or uniqueness of attribute values identifying class instances is the set of instances of this class participating in a containment relationship with the same container instance as this class does.

property_in_container

This value indicates that the scope or uniqueness of attribute values identifying class instances is the set of instances of this class participating in a containment relationship with the same container instance and the same containment relationship as this class does.

Example:

Typical identifier for a UML class contained in a UML Package:

- MyPackage.MyClass, if 'container' is used.
- MyPackage.ownedElement.MyClass, if 'property_in_container' is used

5.1.3 «datatype» ClassRef

This is an AliasType to string, for representing a MOF Class by its fully-qualified name. It is used by ClassConfig (see Section 5.1.1).

taggedValue org.omg.uml2mof.corbaType: string

This type aliases string.

5.1.4 «datatype» AttributeRef

This is an AliasType to string, for representing a MOF Attribute by its fully-qualified name. It is used by IdentifierConfig (see Section 5.1.6), EnumAdjectiveConfig (see Section 5.1.7), DefaultValueConfig (see Section 5.1.8) and ParametricConfig (see Section 5.1.9).

taggedValue org.omg.uml2mof.corbaType: string

This type aliases string.

5.1.5 «datatype» ModelElementRef

This is an AliasType to string, for representing a MOF Model Element by its fully-qualified name. It is used by IdentifierConfig (), Enu

taggedValue org.omg.uml2mof.corbaType: string

This type aliases string.

5.1.6 IdentifierConfig

The metaclass IndentifierConfig is subtype of ClassConfig, which identifies the MOF class being configured for a HUTN language.

The purpose of IndentifierConfig is to nominate a particular attribute of the Class as unique within some scope, so that its value may be used as a unique identifier for the Class in the HUTN language with this configuration.

id_attribute : AttributeRef

This metaattribute refers to the Attribute of the MOF class being configured for a HUTN language representation. If this attribute is null, then arbitrary strings may be used for identifying instances. If this attribute is non null, a specific instance of this class may use or may not use the value of the identifying attribute as the identifier. If the latter case, both the arbitrary identifier and the identifying attribute value should be provided within the HUTN representation of the instance.

uniqueness : UniquenessScope

This metaattribute indicates the scope over which the values of the Attribute being nominated as a unique identifier must be unique. (See section 5.1.2 for the definition of UniquenessScope.)

5.1.7 EnumAdjectiveConfig

The metaclass EnumAdjectiveConfig is subtype of ClassConfig, which identifies the MOF class being configured for a HUTN language.

The purpose of EnumAdjectiveConfig is to identify an attribute of the class whose value may appear as an adjective for instances of this class.

adjectives: set[0..*] of AttributeRef

This metaattribute denotes the MOF Attributes of the Class being configured for a HUTN language which may be used as adjectives.

5.1.8 DefaultValueConfig

The metaclass DefaultValueConfig is subtype of ClassConfig, which identifies the MOF class being configured for a HUTN language.

The purpose of DefaultValueConfig is to provide a value which will be assumed to be the value of the attribute identified when it is not explicitly provided by a class instance declaration.

the_attribute : AttributeRef

This metaattribute refers to the MOF Attribute of the Class being configured for a HUTN language.

the_value : any

This metaattribute provides the default value that the HUTN tool will associate with the attribute for any class instance where it is not given an explicit value.

5.1.9 ParametricConfig

The metaclass ParametricConfig is subtype of ClassConfig, which identifies the MOF class being configured for a HUTN language.

The purpose of ParametricConfig is to provide a list of attributes whose values will be expected in parentheses after the class instance identifier.

parameters : list[0..*] AttributeRef

An ordered list of Attributes for this class configuration which will be placed in parametric form for this HUTN language.

5.1.10 RenameConfig

This metaclass is used to indicate that a Model Element in the MOF Model for which a HUTN language is being configured will take a different name in the HUTN language from its MOF Element Name.

the_element : ModelElementRef

This metaattribute is a reference to the MOF Model Element being renamed in the HUTN configuration.

new_name: string

This metaattribute is the string to be used to represent the_element in the HUTN language.

HUTN Document Production 6

This chapter describes the syntax of the generated languages, in terms of the MOF modeling concepts as outlined in Section 4.2, “The Base Language,” on page 4-1.

The examples presented throughout this chapter (with the exception of the name-scope reduction examples) are derived from the FamilyPackage system, whose information model is described in Section 2.6, “Example XMI,” on page 2-7, and reproduced in Figure 6-1. Section 2.7, “Equivalent HUTN,” on page 2-10 and Section 2.8, “Summary,” on page 2-12 contain the respective XMI and HUTN streams from which the data in this section’s examples are extracted.

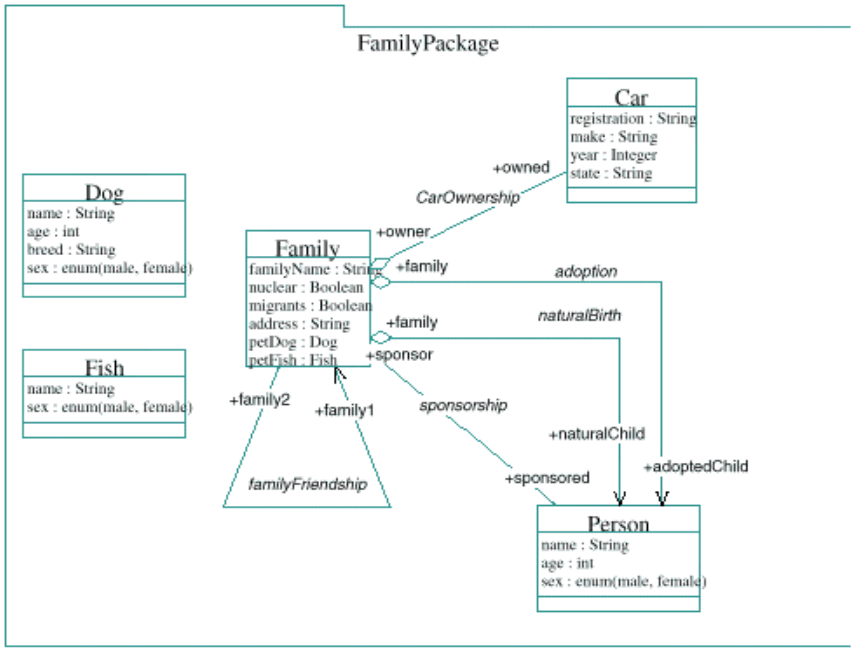

Figure 6-1 MOF model for a family

6.1 Notation

In this chapter, the following notational conventions will be used.

An example of a text stream conforming to a HUTN-generated language will appear as follows. The text in bold face (but not italic) is the literal text stream (e.g., the first and fourth lines below) whereas the text is bold-and-italic face describe omitted detail which should not be taken literally (e.g., the 2nd and 3rd lines below).

FamilyPackage “id-001” {
Class instances here
Association instances or links here
}

EBNF rules for HUTN mappings are presented using a numbered rule for each non-terminal (e.g., PackageInstance, PackageHeader, PackageBody) in the grammar and literal symbols enclosed in single quotes (e.g., right curly brace represented as ‘}’). The italicized words enclosed in angle brackets indicate a placeholder for a literal value which must be substituted with an actual value (e.g., <PackageName>); the name given is deliberately chosen to be meaningful, but will always be explained more fully in the accompanying text.

[2] PackageInstance := 3:PackageHeader ‘{‘ 4:PackageBody ‘}’
[3] PackageHeader := *<PackageName>* 5:PackageIdentifier

6.2 Package Representations

A HUTN document consists of a zero or more instances of the package from which the HUTN has been generated.

[1] Document := (2:PackageInstance)*

A *package* in the MOF type structure is a concept used for containing a collection of related classes and associations.

[2] PackageInstance := 3:PackageHeader
(‘{‘ 4:PackageBody ‘}’
| ‘;’ 4:PackageBody)
[3] PackageHeader := *<PackageName>* 5:PackageIdentifier
[4] PackageBody := (6:ClassInstance | 32:AssocInstance |
24:ClassifierLvlAttr) *
[5] PackageIdentifier := 28:TextualValue

Package instances are represented either by a block structure, with the package contents appearing either between braces, or following a single line introduction followed by a semicolon. Identifying attributes are not permitted on packages and, as such, packages are prefaced and identified by the name of the package, followed by a string-delimited arbitrary unique identifier. This identifier can be used for qualifying

the identifiers of objects referenced between separate package instances in the same document. The package body consists of the class and association instances of the package, as well as any classifier-level attribute values, in accordance with the mappings described in Section 6.3, “Class Representations,” on page 6-3, Section 6.8, “Association Representations,” on page 6-13. An example of the representation of a package instance is given in Figure 6-2, and Section 6.6, “Classifier-Level Attributes,” on page 6-10, respectively.

```
FamilyPackage “id-001” {
    Class instances here
    Association instances or links here
}
```

Figure 6-2 An example of a package instance representation

The classes and associations whose instances can appear in the package body consist of all non-abstract classes and associations that reside in the target package, including those that result from package inheritance, those that are defined in nested packages, and those that are defined in clustered packages.

6.3 Class Representations

```
[6]   ClassInstance     := 7:ClassHeader 10:ParametricAttrs
                           ‘{‘ 11:ClassContents ‘}’ (‘;’)?
[7]   ClassHeader       := 8:ClassAdjectives <ClassName>
                           (9:ClassIdentifier)?
[8]   ClassAdjectives   := ( (‘~’? <AttributeName>) | 25:DataValue) *
[9]   ClassIdentifier   := 28:TextualValue
[10]  ParametricAttrs   := ‘(‘ 31:ValueList ‘)’
[11]  ClassContents     := (  (18:AttributeInstance
                              | 21:ReferenceInstance
                              | 12:ContainedObject
                             )  (‘;’)?
                           ) *
[12]  ContainedObject   := (<AssocName>’:’)?
                           (6:ClassInstance | 13:ClassInstanceRef)
```

The representation of a class consists of a number of parts; adjectives, class name, identifier, parametric representation, and contents. The contents consist of attribute, reference and contained object representations. The parts appear in that order, to form the representation of the class. These attribute, reference and contained instance representations appear within curly braces, in any order, optionally terminated by semicolons.

There are two types of adjectives: single-valued boolean attributes, for which adjective representation is enabled by default; and single-valued enumeration-typed attributes, which must be configured as adjectives. Boolean attributes are represented by the

attribute name, optionally prefixed by the '~' symbol, representing negation. Enumerated adjectives are represented as the enumerator label corresponding to the value of the attribute. Adjectives can appear in any order.

Class instances can be identified in one of two ways. Firstly, if a single-valued simple-typed attribute has been configured as the class' identifying attribute, then the value of that attribute, formatted as appropriate for the attribute's type, appears as the identifier. Alternatively, if no such identifier has been selected, then an arbitrary string may be used, or the instance may go unidentified. Unidentified instance representations have no identifier, and may only be used when the instance is not referred to from anywhere else in the document.

If any single-valued, simple-typed attributes have been configured for parametric representation, then their values appear next inside parentheses. The values appear in the order specified by the parametric configuration, and are separated by whitespace or commas.

Contained objects are those class instances linked by a containment association for which there are no references defined.

Figure 6-3 and Figure 6-4 show examples of class representations, where the Family class is identified arbitrarily and by the familyName attribute, respectively. In both examples, the boolean nuclear attribute is used as an adjective.

```
FamilyPackage "id-001" {
    Family "id-002" {
        familyName: "The McDonalds"
        Attribute and Reference representations
}
nuclear Family "id-003" {
    familyName: "The Smiths"
        Attribute and Reference representations
    }
}
```

Figure 6-3 An example of the representation of arbitrarily-identified class instances

```
FamilyPackage "id-001" {
    Family "The McDonalds" {
        Attribute and Reference representations
}
nuclear Family "The Smiths" {
        Attribute and Reference representations
    }
}
```

Figure 6-4 An example of the representation of attribute-identified Class instances

Figure 6-5 shows an example for metamodel of polygons, with a string attribute for their name, a boolean attribute for whether they are filled or not, and containment association with coordinate class. The coordinate class has only two floating point attributes, X and Y. The HUTN configuration for coordinates places X and Y in parametric form, and for polygons it provides the name attribute as a unique identifier and the default value ‘true’ for the filled attribute. The example shows filled used as a negated boolean adjective and the instance identifier as an undelimited string. The contained coordinate objects are shown inline with all their attribute values in parametric form, and their empty contents denoted by the use of a semicolon.

```
ShapePackage triangles {
    ~filled polygon my_triangle {
        coordinate (3.6, 7.3);
        coordinate (5.2, 7.673);
        coordinate (9.4 ,13);
    }
}
```

Figure 6-5 An example of the use of various configuration options

```
[13]    ClassInstanceRef  := ( (<ClassName>)? 14:ClassRefString )
                             | ExternalObjRef
[14]    ClassRefString    := 15:PackageRootRef | 16:DocumentRootRef
[15]    PackageRootRef    := 17:ClassRefSeparator  28:TextualValue
                             (17:ClassRefSeparator 28:TextualValue) *
[16]    DocumentRootRef := 17:ClassRefSeparator 17:ClassRefSeparator
                             28:TextualValue
                             (17:ClassRefSeparator 28:TextualValue) *
[17]    ClassRefSeparator:= ‘::’ | ‘.’ | ‘/’
```

There are number of cases in which a class instance can be referred to, either as a non-contained attribute of another class instance, via a MOF Reference, or in an association. The standard representation of one of these references consists of the class name followed by the identifying string for the class instance (explained below). Alternatively, the typeless references shorthand allows for the omission of the type name of the referred instance, leaving just the identifying string. (This is subject to the conditions stated in Section 4.2.3, “Defining and referencing major concepts,” on page 4-2).

The string used to refer to class instances is structured differently according to the uniqueness scope of the referred class’ identification system. If all_of_type scope is used, then the string is just the class identifier of the class referred to (arbitrary or identifying-attribute value). If container scope is used, then a number of these strings can be separated by either a double-colon, a full-stop, or a forwards-slash. These names are resolved relatively, moving up the containment hierarchy one level at a time until a match is found. If container is used then a level is represented by a single string,

which is the container's identifier. If property_in_container is used then a level is represented by two strings separated by a delimiter: the container's identifier and the property name of the containment association. To indicate that the name should be resolved relative to the current package instance, a separator can be included as a prefix to the string. Regardless of which uniqueness scope is used, instances in other packages can be referenced by prefixing the string with a double separator followed by the identifying string of the package instance that contains the referred object.

If the class instance referred to exists outside of the scope of the current document (including any references to external imported classes), then it is represented according to the rules in Section 6.7.5, "Object Reference," on page 6-11.

Figure 6-6 shows an further example for the metamodel of polygons with an additional class "diagram," which has a string name and a multi-valued-attribute "shapes" of type polygon. The text below is assumed to be in the same document as Figure 6-5.

```
ShapePackage quads {
    polygon my_quad1 {
        coordinate (4.6, 78.3);
        coordinate (4.2, 7.3);
        coordinate (10.4 ,1.5);
        coordinate (33 ,8.5);
    }

    diagram two_shapes {
        shapes = [ //triangles/my_triangle, /my_quad1 ]
    }
}
```

Figure 6-6 An example of the use of package and document references

6.4 Attribute Representations

[18] AttributeInstance := 19:NormalAttribute | 20:KeywordAttribute
[19] NormalAttribute := <*AttributeName*> (':' | '=')
(25:DataValue | 'null')
[20] KeywordAttribute := ('~')? <*AttributeName*>?

The standard representation for attributes consists of the attribute name, followed by a colon or by an equals sign '=', followed by the data value of the attribute, encoded as is appropriate for the attribute's type. There are, however, a number of shorthands for attribute representation. If a default value has been specified for the attribute, then the absence of the attribute's representation must be taken to mean that the attribute has the default value. Mandatory boolean attributes can be represented using the adjective or keyword shorthands. If there is no default value configured then a value of true is indicated by the attribute name, and a value of false by the absence of the attribute name. The same applies if a default value of false has been configured. However, if a

default value of true is configured then the absence of the attribute name indicates a value of true, and a value of false is represented by the attribute name preceded by a tilde '~'.

Attributes whose lower multiplicity bound is 0 may be explicitly unset by assigning them to the 'null' keyword.

Figure 6-7 presents an example of a number of attributes' representations. The 'migrants' attribute has been used as a keyword on the Family class, the 'nuclear' attribute as an adjective of Family, 'familyName' configured the identifying attribute of Family, and 'name' has been configured as the identifier of Person.

```
FamilyPackage "id-001" {
    Family "The McDonalds" {
        migrants
        Address: "7 Main Street"
        Reference representations
    }
    nuclear Family "The Smiths" {
        Address: "5 Main Street"
        Reference representations
    }
    Person "Namdou Ndiaye" {
        age: 7
        sex: male
        Reference representations
    }
}
```

Figure 6-7 An example of representations of simple attributes

Attributes whose values are instances of a class are represented in two separate ways. If the attribute class instance is contained by the enclosing class instance (that is, it does not exist outside of the containing instance's scope), then the attribute instance may represented in-line in the manner described in Section 6.3, "Class Representations," on page 6-3. Alternatively, a class instance may appear as a referred object, as described in Section 6.3, "Class Representations," on page 6-3. If the class instance is not contained, then only this second representation may be used. An example in which petFish is a contained attribute and petDog is a non-contained attribute, both of Family, is presented in Figure 6-8.

```
FamilyPackage “id-001” {
    Family “The McDonalds” {
        petDog: Dog “Spike”
        petFish: Fish “Wanda” {
            Attribute and reference representations
        }
    }
    Dog “Spike” {
            Attribute and reference representations
    }
}
```

Figure 6-8 An example of class-instance valued attributes

6.5 Reference Representations

```
[21]  ReferenceInstance     := 22:ContainedReference
                               | 23:NonContReference
[22]  ContainedReference    := ( <ReferenceName> (':' | '=') )?
                               (6:ClassInstance | 13:ClassInstanceRef)
[23]  NonContReference      := <ReferenceName> (':' | '=')
                               13:ClassInstanceRef
```

References are a means for classes to be aware of class instances that play a part in an association, by providing a view into the association as it pertains to the observing instance. For this reason, the representation within a class instance of a reference depends in part on the nature of the association to which it refers. An association is involved in a *containment relationship* if one of the participating instances is wholly contained within the other. That is, the *contained instance* does not exist outside the scope of the other instance.

Much like that of an attribute, the representation of a reference begins with the name of the reference followed by a colon or an equals sign. If the instance to which the reference refers is the contained instance in a containment relationship, then this may be followed either by a full representation of the instance (see Section 6.3, “Class Representations,” on page 6-3), or by a reference to the instance. In the latter case the full representation of the contained instance must appear as a top level definition in the content of the current package. Figure 6-9 shows the Family class with references, ‘naturalChild’ and ‘adoptedChild’, to two containment associations between the Family and Person classes.

```
FamilyPackage “id-001” {
    Family “The Smiths” {
        Attribute representations
        Reference representations

        naturalChild: Person “Harry Smith” {
            Attribute and reference representations
        }
        naturalChild: Person “Joan Smith” {
            Attribute and reference representations
        }
        adoptedChild: Person “Dylan Smith” {
            Attribute and reference representations
        }
    }
}
```

Figure 6-9 An example representation for a reference to a containment association

If there is only one association through which a contained object may be referred, then the shorthand of a nameless reference is available, in which the name of the reference (and the trailing colon) may be omitted.

Alternatively, if the association that is referred to is not a containment relationship, then the subsequent depiction must be in the form of an instance reference. An example of this case is given in Figure 6-10, where familyFriends is a reference to a non-containment association ‘familyFriendship’.

```
FamilyPackage “id-001” {
    Family “The McDonalds” {
        Attribute representations
        familyFriends: Family “The Smiths”
    }
    Family “The Smiths” {
        Attribute representations
        familyFriends: Family “The McDonalds”
    }
}
```

Figure 6-10 An example of the representation of references to a non-containment association

If the class referred to by a non-containment association (and all its subclasses) use a common identification mechanism (either a single identifying attribute or the arbitrary unique identifier), then the type name of the referred class may be omitted, as a typeless reference.

It should be noted that an association link need only be represented once throughout the document. For example, in the case of an association between two classes where both classes have references to the association, a link need only be shown in one of the three possible places it may appear; in one of the two references, or in the representation of the association. The link may be shown more than once, so long as the different representations are consistent.

6.6 Classifier-Level Attributes

[24] ClassifierLvlAttr := *<ClassifierName>* '.' *<AttrName>* (':'|'=') 25:DataValue ';'

Classifier-level Attributes are represented similarly to other attributes, with the exception that their declarations must appear within package bodies rather than within class instance bodies, and that the Attribute name must be prefixed by the name of the Classifier. These declarations must be terminated by semi-colons.

6.7 Data Value Representations

[25] DataValue := 26:SingleValueData | 30:MultiValueData
[26] SingleValueData := 28:TextualValue
| NumericValue
| EnumValue
| 27:BooleanValue
| TypeCodeValue
| StructValue
| UnionValue
| 6:ClassInstance
| 13:ClassInstanceRef

The data types of Attributes in MOF are based on CORBA TypeCodes. Therefore, for each concrete TypeCode, the HUTN must define a textual representation for instances of that type. Furthermore, a MOF Attribute may be defined as a collection kind (Set, Bag, List, UList) of these data types.

Note that the system of data typing used by MOF is currently being revised, and that a final submission will need to conform to these revised data types.

6.7.1 Numeric types

Shorts, longs, unsigned shorts, unsigned longs, floats, doubles, longlongs, unsigned longlongs, long doubles, fixed points and octets are all represented as numeric literals (see Section 6.9.5, "Numeric literals," on page 6-15).

6.7.2 Boolean

[27] BooleanValue := 'true' | 'false'

Boolean values, true and false, are represented as reserved words (see Section 6.9.3, "Reserved Words," on page 6-15). Note that boolean attributes may also be represented as keywords or adjectives, which appear in the class header.

6.7.3 Textual types

[28] TextualValue := *EncodedString*

Characters, strings, wide characters and wide strings are represented as literal strings (see Section 6.9.6, "Character and string literals," on page 6-16). The encoded string may be delimited by either double quotes, single quotes, or back quotes. Alternatively, they may go undelimited, provided that they start with an alphabetic character, and that they contain no whitespace or special characters.

6.7.4 Enum

An enum-value is represented as an identifier (see Section 6.9.2, "Identifiers," on page 6-15) with the string values being the names of the enum labels. Note that attributes of type enum may also be presented as adjectives, which appear in the same way.

6.7.5 Object Reference

[29] ExternalObjRef := *StringifiedObjRef*

An object-reference value is represented as a string literal. The format of the string is defined by the CORBA standard (Sections 13.6.6 through 13.6.7). It should be noted that some of the URL formats defined in 13.6.7 were designed to be more "human-usable" than the stringified object reference format of Section 13.6.6 (a stream of hexadecimal digits). However, the stringified object reference may be the only format that can be generated by some ORBs (using the operation object_to_string).

It is important to note that these object-reference values are NOT the format used to cross-reference objects within a HUTN document, as these are instances of MOF Classes and are represented by their class name followed by their object identifier (see Section 6.3, "Class Representations," on page 6-3). These object-reference values are used to reference external CORBA objects.

6.7.6 TypeCode

A TypeCode-value is represented as a literal string (see Section 6.9.6, "Character and string literals," on page 6-16). The format of the string is defined in XMI 1.2 Sections 6.4.8.2 through 6.4.8.16.

6.7.7 Any

An any-value is represented as a bracketed pair (see Section 6.9.7, “Bracketed Pairs/Lists,” on page 6-16). The first value is the TypeCode (see Section 6.7.6, “TypeCode,” on page 6-11) and second is the data value (see Section 6.7, “Data Value Representations,” on page 6-10) which is represented appropriately for the stated TypeCode. The pair are separated by white space and grouped with brackets (either round, square, or angle).

6.7.8 Struct

A struct-value is represented as a bracketed list (see Section 6.9.7, “Bracketed Pairs/Lists,” on page 6-16), containing the values of each of the fields of the struct in the order defined by the struct. The fields of the structs are not labelled.

6.7.9 Union

A union-value is represented as a bracketed list (see Section 6.9.7, “Bracketed Pairs/Lists,” on page 6-16). The first value is the value of the discriminator. The second value is the value of the variant part selected by the discriminator (if any).

6.7.10 Sequence, Array

A sequence-value or an array-value is represented as a bracketed list (see Section 6.9.7, “Bracketed Pairs/Lists,” on page 6-16) with each value of the sequence/array represented in the same order as in the sequence/array.

6.7.11 Collections (Set, Bag, List, UList)

A collection value is represented as a bracketed list (see Section 6.9.7, “Bracketed Pairs/Lists,” on page 6-16) with each value of the collection represented exactly once. For ordered collections (Lists and ULists), the elements of the bracketed list are in the same order as the collection. For unordered collections (Sets and Bags), the ordering does not matter.

```
[30]    MultiValueData     := '<' 31:ValueList '>'
                            | '[' 31:ValueList ']'
                            | '(' 31:ValueList ')'
[31]    ValueList          := (25:DataValue)+
                            |  25:DataValue (',' 25:DataValue)*
```

An alternative representation is to provide a number of attribute name-value pairs, one for each value in the collection.

6.8 Association Representations

Associations constitute a relationship between two classes, and can appear in two forms: either containment relationships or non-containment relationships. Further to this, classes can contain references into associations (see Section 6.5, “Reference Representations,” on page 6-8). This leads to three methods of representing the link between associated class instances.

Firstly, if one or more of the classes participating in the association contains a reference into the association, then the elements participating in the association can displayed within the representation of the class containing the reference, as described above in Section 6.5, “Reference Representations,” on page 6-8.

Secondly, if the association represents a containment relationship, but there are no classes with references to the association, the association contents may be displayed within the representations of the containing class instances. The representation for the contained instance is exactly the same as if it were referenced, except that the name of the association is substituted for the name of the reference. This is shown by the production rules in Section 6.3, “Class Representations,” on page 6-3. The association name is optionally displayed before the contained class instance to allow disambiguation for MOF models which have more than one possible containment association between the container and the contained instance.

An example of the representation of unreferenced containment associations is presented in Figure 6-11, where CarOwnership is an association between Family and Car, with Car instances being contained by Family instances.

```
FamilyPackage “id-001” {
    Family “The McDonalds” {
        CarOwnership: Car “755-BDL” {
            Attribute and reference representations
        }
    }
}
```

Figure 6-11 An example of a containment association without references

```
[32]    AssocInstance     := 33:AssocBlock | 36:InfixAssocLink
[33]    AssocBlock        :=<AssocName> ‘{‘ 34:AssocContents ‘}’
[34]    AssocContents     := ( 35:AssocEnd 35:AssocEnd ) *
[35]    AssocEnd          := (<AssocEndName> (’:’ | ‘=’) )?
                             13:ClassInstanceRef
```

The third method of representing association instances involves displaying the link separately to either of the class instances that participate. There are two forms for this: block display of the association, or infix representation of the individual links.

In the case of block display, a list appears containing references to the class instances participating in the association. The block consists of the name of the association followed by a block (with opening and closing braces) containing the pairs of references to the instances participating in the relationships. Each instance in the pair

may optionally be preceded by the name of the role it plays in the association and a colon or equals symbol. Class instance references are displayed in the style specified in Section 6.3, “Class Representations,” on page 6-3. Figure 6-12 shows a block representation of an association ‘sponsorship’ between the Family and Person classes..

```
FamilyPackage “id-001” {
    Family “The Smiths” {
        Attribute and reference representations
    }
    Person “Namdou Ndiaye” {
        Attribute and reference representations
    }
    sponsorship {
        sponsor: Family “The Smiths”
        sponsored: Person “Namdou Ndiaye”
        Other pairs within the sponsorship association
    }
}
```

Figure 6-12 An example of the representation for a non-containment association

[36] InfixAssocLink := 13:ClassInstanceRef *<AssocName>*
13:ClassInstanceRef

Infix display consists of references to each of the class instances (in the form appropriate for the participating classes’ identification configuration), separated by the name of the association. It should be noted that the ends of associations are ordered, and the participating class instances must appear in the appropriate order, with the first association end before the association name and the second afterwards. Infix links may be optionally terminated by a semicolon. An example of infix display is shown in Figure 6-13, where Family is the first end of the sponsorship association, and Person is the other end...

```
Family “The Smiths” sponsorship Person “Namdou Ndiaye”;
```

Figure 6-13 An example of infix display for associations

As mentioned in Section 6.5, “Reference Representations,” on page 6-8, there need only be one representation of any an association link between a pair of class instances. This can be in the form of a reference, or using any of the forms shown above for associations.

6.9 Lexical issues

In most lexical aspects, HUTN follows the accepted practices of OMG IDL, but allows greater freedom where the strong-typing of the underlying model permits it.

6.9.1 Comments

Comments appear as in OMG IDL:
/* comment between delimiters */
// comment to end of line

A leading or 'header' comment in a document is treated specially, in that it may contain a HUTN configuration for the document, or a URL or URI indicating where the configuration document can be found. See Chapter 7 for the specification of the HUTN Configuration for the metamodel given in Chapter 5.

These comments take the following form:

```
/**
  *@config
   HUTN Configuration document text  or URL/URI
*/
```

6.9.2 Identifiers

An identifier is an arbitrarily long sequence of alphabetic, numeric, and underscore characters. The first character must be alphabetic. Identifiers in HUTN-generated languages are mostly taken from the names of MOF Packages, Classes, Attributes, References, and Associations. Some identifiers from the MOF model will be replaced by renaming configurations.

Identifiers are case-sensitive, and there are no clashing-case rules like those in OMG IDL.

6.9.3 Reserved Words

The HUTN languages have three reserved words: "true" and "false" for representing boolean values, and "null" for unsetting attribute values.

6.9.4 White Space

Like OMG IDL, white space and comments can be freely used between lexical elements (but not within them) and are ignored in parsing. Note that white space or comments must be used to separate lexical elements such as identifiers and reserved words (which would otherwise appear as a single lexical element without such separation).

6.9.5 Numeric literals

The legitimate forms for numeric literals (integers, floating point, and fixed point) are the same as for OMG IDL.

'+' and '-' can be used to indicate the positive/negative sign of the value.

It should be noted that OMG IDL permits the presentation of integer literals in octal and hexadecimal forms using the prefixes 0 and 0x/0X respectively.

6.9.6 Character and string literals

The contents of character and string literals can take any legitimate form defined by OMG IDL. Unlike OMG IDL, however, strings may be delimited by either single quotes, double quotes, or back quotes (provided that the opening delimiter matches the closing). Strings may also be left undelimited, provided that they start with an alphabetic character, and contain no whitespace or special characters. Characters are delimited by matching single, double or back quotes. Escape sequences are supported in both strings and characters. Wide characters/strings are prefixed with “L.” String concatenation is supported (primarily to enable long strings to be entered using a number of lines).

Since HUTN is based on the strong type system of the MOF, it is always known whether a literal value is a character or a string. Hence, HUTN permits the use of a pair of single, double or back quotes to delimit both character and string literals. This can be convenient when the string must contain single/double quote characters as it is only necessary to escape the kind of quote used as a delimiter.

6.9.7 Bracketed Pairs/Lists

Bracketed pairs/lists appear throughout the generated HUTN grammars. They consist of 0 or more (strictly 2 for a pair) values separated by white space or commas, surrounded by matching brackets. The brackets can be either square [], round (), or angle <>.

As a number of the representations that use bracketed pairs/lists are recursively defined, it is to be expected that bracketed pairs/lists will be nested. The use of different kinds of brackets may help to make the groupings clearer in the text when there is extensive nesting.

Bracketed form is also used for parametric form, but in this case the brackets are always round, and there is at least one value in the list (See Section 6.3, “Class Representations,” on page 6-3).

6.9.8 Symbols

The generated HUTN grammars use the following symbols in a consistent manner as described in Table 6-1:

Table 6-1 Use of symbols in HUTN-generated languages

Symbol	Symbol name	Use in HUTN
{}	Curly braces	nesting of content
( )	Round brackets	grouping of parametric form

Table 6-1 Use of symbols in HUTN-generated languages

( )	Round brackets	grouping of lists
[ ]	Square brackets	
<>	Angle brackets	
:	colon	introduces a data value
=	equals sign	
+	plus	sign of number
-	minus	
'	single quote	delimit literal strings
‘	back quote	
“	double quote	
\	backslash	escape in literal strings
,	comma	separator in bracketed lists and parametric attributes
~	tilde	boolean adjective negation
;	semi-colon	optional terminator of attributes, references, and association links

6.10 Name Scope Optimization

Names of packages, associations, and classes in the MOF include all of the information about the concept’s scope. This fully qualified name consists of a number of scope-level components, separated by dots. For example, an attribute contains information about which class it is in, and what package that class is contained by. However, while this scope information is necessary in the broader picture, these names provide more information than is necessary to uniquely identify a model concept within the model.

The names of packages, associations, and classes are therefore optimized to make them as short as possible while still being unique within the domain model. (Since attribute names are unique within their class, they are simply represented by their local name). This is done as follows. First, a set of all names is assembled, and each is broken down into a sequence of words (one for each scope level). A possible scoped name is then created for each name, constituting the last word of the word sequence for that name. If this possible name is unique within the set of possible names, then it is accepted as the scope-optimized name. If not, then the process is repeated with the last two words of the name sequence. This continues until all names have been optimized. The table shown in presents an example of a set of names and their reductions.

Table 6-2 An example of some name optimizations

Fully Scoped Name	**Scope-Optimized Name**

Table 6-2 An example of some name optimizations

Genealogy.Family.Child	Family.Child
Genealogy.Family.Father	Father
Genealogy.Tree.Child	Tree.Child
Genealogy.Tree.Branch	Genealogy.Tree.Branch
Flora.Tree.Branch	Flora.Tree.Branch
Flora.Flower	Flower

Configuration Notation 7

HUTN language configurations are expressed using a HutnConfig language generated according to the rules in Chapter 6. The language configuration for this generated notation is described in this chapter.

7.1 HutnConfig Language Configuration

The following document is the language configuration of the HUTN language for the HutnConfig MOF metamodel. The document is configured by itself, and therefore the body text is duplicated in the @config section of the opening comment. This demonstrates the use of the @config statement for specifying language configurations within HUTN documents.

```
/**
 * @config
    HutnConfig “HutnConfig” {
        all_of_type IdentifierConfig “HutnConfig.IdentifierConfig” {
            id_attribute: “HutnConfig.ClassConfig.the_class”
        }
        EnumAdjectiveConfig “HutnConfig.IdentifierConfig” {
            adjectives: “HutnConfig.IdentifierConfig.uniqueness”
        }
        all_of_type IdentifierConfig “HutnConfig.EnumAdjectiveConfig” {
            id_attribute: “HutnConfig.ClassConfig.the_class”
        }
        all_of_type IdentifierConfig “HutnConfig.ParametricConfig” {
            id_attribute: “HutnConfig.ClassConfig.the_class”
        }
        all_of_type IdentifierConfig “HutnConfig.RenameConfig” {
            id_attribute: “HutnConfig.RenameConfig.the_element”
        }
    }
 */
```

```
HutnConfig “HutnConfig” {
    all_of_type IdentifierConfig “HutnConfig.IdentifierConfig” {
        id_attribute: “HutnConfig.ClassConfig.the_class”
    }
    EnumAdjectiveConfig “HutnConfig.IdentifierConfig” {
        adjectives: “HutnConfig.IdentifierConfig.uniqueness”
    }
    all_of_type IdentifierConfig “HutnConfig.EnumAdjectiveConfig” {
        id_attribute: “HutnConfig.ClassConfig.the_class”
    }
    all_of_type IdentifierConfig “HutnConfig.ParametricConfig” {
        id_attribute: “HutnConfig.ClassConfig.the_class”
    }
    all_of_type IdentifierConfig “HutnConfig.RenameConfig” {
        id_attribute: “HutnConfig.RenameConfig.the_element”
    }
}
```

ECA Textual Notation 8

This chapter specifies a HUTN language configuration for the ECA metamodel as specified in the UML Profile for EDOC adopted specification (ptc/2002-02-05). This configuration, in conjunction with the ECA metamodel as specified in the above document, when used with the production rules for Human-Usable Textual Notations, result in a textual notation for the expression of ECA models.

8.1 ECA Language Configuration

The following is the language configuration for the ECA metamodel.

```
HutnConfig “org.omg.ECA” {

    // ModelManagement package configurations
    all_of_type IdentifierConfig
            org.omg.ECA.ModelManagement.PackageContent {
        id_attribute:
            “org.omg.ECA.ModelManagement.PackageContent.name”
    }

    // DocumentModel package configurations
    all_of_type IdentifierConfig org.omg.ECA.DocumentModel.DataInvariant
    {
        id_attribute:
            “org.omg.ECA.DocumentModel.DataInvariant.expression”
    }
    container IdentifierConfig
            org.omg.ECA.DocumentModel.EnumerationValue {
        id_attribute: org.omg.ECA.DocumentModel.EnumerationValue.name
    }
    DefaultValueConfig {
        the_class: org.omg.ECA.DocumentModel.ECAAttribute
```

```
        the_attribute:
            org.omg.ECA.DocumentModel.ECAAttribute.initialValue
        the_value: ""
}

// CCA package configurations
container IdentifierConfig org.omg.ECA.CCA.Node {
    id_attribute: org.omg.ECA.CCA.Node.name
}
EnumAdjectiveConfig org.omg.ECA.CCA.PseudoState {
    adjectives: org.omg.ECA.CCA.PseudoState.kind
}
container IdentifierConfig org.omg.ECA.CCA.ComponentUsage {
    id_attribute: org.omg.ECA.CCA.ComponentUsage.name
}
container IdentifierConfig org.omg.ECA.CCA.PropertyValue {
    id_attribute: org.omg.ECA.CCA.PropertyValue.value
}
container IdentifierConfig org.omg.ECA.CCA.InitiatingRole {
    id_attribute: org.omg.ECA.CCA.InitiatingRole.name
}
container IdentifierConfig org.omg.ECA.CCA.RespondingRole {
    id_attribute: org.omg.ECA.CCA.RespondingRole.name
}
container IdentifierConfig org.omg.ECA.CCA.Port {
    id_attribute: org.omg.ECA.CCA.Port.name
}
EnumAdjectiveConfig org.omg.ECA.CCA.Port {
    adjectives: org.omg.ECA.CCA.Port.direction
}
EnumAdjectiveConfig org.omg.ECA.CCA.ProcessComponent {
    adjectives: org.omg.ECA.CCA.ProcessComponent.granularity
}
DefaultValueConfig {
    the_class: org.omg.ECA.CCA.ProcessComponent
    the_attribute: org.omg.ECA.CCA.ProcessComponent.primitiveKind
    the_value: ""
}
DefaultValueConfig {
    the_class: org.omg.ECA.CCA.ProcessComponent
    the_attribute: org.omg.ECA.CCA.ProcessComponent.primitiveSpec
    the_value: ""
}
container IdentifierConfig org.omg.ECA.CCA.PropertyDefinition {
    id_attribute: org.omg.ECA.CCA.PropertyDefinition.name
}
DefaultValueConfig {
    the_class: org.omg.ECA.CCA.PropertyDefinition
    the_attribute: org.omg.ECA.CCA.PropertyDefinition.initial
    the_value: ""
}
```

```
// Event package configurations
container IdentifierConfig org.omg.ECA.Event.EventCondition {
    id_attribute: org.omg.ECA.Event.EventCondition.condition
}
container IdentifierConfig org.omg.ECA.Event.NotificationRule {
    id_attribute: org.omg.ECA.Event.EventCondition.condition
}
container IdentifierConfig org.omg.ECA.Event.Subscription {
    id_attribute: org.omg.ECA.Event.subscriptionClause
}
DefaultValueConfig {
    the_class: org.omg.ECA.Event.Subscription
    the_attribute: org.omg.ECA.Event.Subscription.domain
    the_value: ""
}
container IdentifierConfig org.omg.ECA.Event.Subscription {
    id_attribute: org.omg.ECA.Event.subscriptionClause
}
DefaultValueConfig {
    the_class: org.omg.ECA.Event.Publication
    the_attribute: org.omg.ECA.Event.Publication.domain
    the_value: ""
}

// BusinessProcessPkg package configurations
DefaultValueConfig {
    the_class: org.omg.ECA.BusinessProcessPkg.ProcessFlowPort
    the_attribute:
    org.omg.ECA.BusinessProcessPkg.ProcessFlowPort.multiplicity_lb
    the_value: 1
}
DefaultValueConfig {
    the_class: org.omg.ECA.BusinessProcessPkg.ProcessFlowPort
    the_attribute:
org.omg.ECA.BusinessProcessPkg.ProcessFlowPort.multiplicity_ub
    the_value: 1
}
DefaultValueConfig {
    the_class: org.omg.ECA.BusinessProcessPkg.ProcessRole
    the_attribute:
        org.omg.ECA.BusinessProcessPkg.ProcessRole.selectionRule
    the_value: ""
}
DefaultValueConfig {
    the_class: org.omg.ECA.BusinessProcessPkg.ProcessRole
    the_attribute:
        org.omg.ECA.BusinessProcessPkg.ProcessRole.creationRule
    the_value: ""
}
}
```

References H

A.1 List of References

[Antlr] *Terence Parr, ANTLR - Complete Language Translation Services. http://www.antlr.org/index.html*

[Belaunde99] *Mariano Belaunde, A Pragmatic Approach To Building a Flexible UML Model Repository. In UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings. Springer. Pp 188-203.*

[Java] *James Gosling, Bill Joy and Guy Steele. "The Java™ Language Specification", First Edition. Sun Microsystems, 1996*

[JavaCC] *Sun Microsystems & Metamata. The Java™ Parser Generator. http://www.metamata.com/javacc/index.html.*

[McIver96] *Linda McIver and Damian Conway. Seven Deadly Sins of Introductory Programming Language Design. In Proceedings, 1996 Conference on Software Engineering: Education and Practice. IEEE Computing Society Press, Los Alamitos, CA, USA 1996. Pp 309-316.*

[MOF01] *Meta-Object Facility (MOF) Specification, OMG TC document formal/2001-11-02, 2001*

[RL77] *Frederic Richard and Henry F. Ledgard. A Reminder for Language Designers. ACM SIGPLAN Notices, Vol. 12 No. 12 (December 1977). Pp 73-82.*

[Visibroker] *Inprise Corporation. "Visibroker 3.4 for Java". http://www.visigenic.com/visibroker/*

[XMI02] *XML-Based Model Interchange (XMI) Specification, OMG TC document formal/2002-01-01, 2002.*

[XML98] *eXtensible Markup Language (XML) 1.0, World Wide Web Consortium Recommendation 10-February-1998. Http://www.w3.org/TR/1998/REC-xml-19980210.*

[XSLT99] *XSL Transformations (XSLT) Version 1.0, W3C Proposed Recommendation 8 October 1999. http://www.w3.org/TR/1999/PR-xslt-19991008.*

[XT99] *James Clark. XT. http://www.jclark.com/xml/xt.html.*