UML™ Profile and Interchange Models for Enterprise Application Integration (EAI) FTF Interim Convenience Document

This OMG document shows changes to the adopted specification (ptc/2002-02-02) prepared by the Finalization Task Force. It is an interim document showing work in progess. It should be read in conjunction with the Interim FFT Report (ptc/2003-02-01).

OMG FTF Interim Convenience Document
ptc/2003-02-02

UML™ Profile and Interchange Models for Enterprise Application Integration (EAI) Specification

Final Adopted Specification
February 2002

Copyright 2001 DSTC
Copyright 2001 Hitachi, Ltd.
Copyright 2001 IBM Corporation
Copyright 2001 Oracle Corporation
Copyright 2001 Rational Corporation
Copyright 2001 Unisys Corporation

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.

PATENT

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.

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR 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 indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. 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. 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.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG® and Object Management are registered trademarks of the Object Management Group, Inc. OMG OBJECT MANAGEMENT GROUP, CORBA, CORBA ACADEMY, CORBA ACADEMY & DESIGN, THE INFORMATION BROKERAGE, OBJECT REQUEST BROKER, OMG IDL, CORBAFACILITIES, CORBASERVICES, CORBANET, CORBAMED, CORBADOMAINS, GIOP, IIOP, OMA, CORBA THE GEEK, UNIFIED MODELING LANGUAGE, UML, and UML CUBE LOGO are registered trademarks or trademarks of the Object Management Group, Inc.

X/Open is a trademark of X/Open Company Ltd.

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 the 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.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows.

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.
- ***Common Warehouse Metamodel (CWM) Specification*** mainly consists of definitions of metamodels in the following domains:
  - Object model (a subset of UML)
  - CWM foundation
  - Relational data resources
  - Record data resources
  - Multidimensional data resources
  - XML data resources
  - Data transformations
  - OLAP (On-line Analytical Processing)
  - Data mining
  - Information visualization
  - Business nomenclature
  - Warehouse process
  - Warehouse operation

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency, Event, Externalization, Naming, Licensing, Life Cycle, Notification, Persistent Object, Property, Query, Relationship, Security, Time, Trader,* and *Transaction*.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility.*

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.

- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Life Science*: Comprised of specifications that relate to the OMG-compliant interfaces for the life science industry.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

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. To obtain print-on-demand books in the documentation set or other OMG publications, 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

Structure of this Document

Part 1 - Introduction

Chapter 1 includes the introductory information, scope, modeling approach and compliance requirements.

Part 2 - Metamodel

Comprised of Chapter 2 (EAI Integration Metamodel and Chapter 3 (EAI Common Application Metamodel).

Part 3 - Profile Definition

The profile presented here focuses on two main modeling approaches, based on collaborations and based on activities. These are described in Chapters 4 and 5, respectively.

The collaboration-modeling approach is based on a modeling framework of classes that provide detailed definitions of the semantics of the collaboration. It is thus useful for providing the detailed specification of message flows in the design of integration subsystems.

The activity-modeling approach is based on the use of activity graphs. This approach is particularly useful for showing the overall control and data flow required for integration, typically at a higher level than in collaboration modeling.

Casting the metamodel as a UML profile allows EAI architecture models to be notated using standard UML notation. This means that most UML tools (specifically ones which support the extension mechanisms of UML, such as stereotypes and tagged values) can be used to define EAI architecture models.

Standard practice for defining UML profiles has been adopted. A mapping of metamodel classes to their base UML classes, with accompanying stereotypes, tagged values and constraints is summarized for each approach. An implementation of this mapping can be used, for example, to generate metadata conforming to the EAI metamodel from XMI generated from models notated using the UML profile. Specialized EAI tools will more likely use the metamodel than the UML profile as a basis for storing and manipulating models.

The art of defining a UML profile is to provide the best fit possible with UML, so that the notation is natural for a modeler in the relevant domain (EAI in this case), and fits with one's general intuitions about the the meaning of the elements of UML that are used in the profile. The profile described here has been designed with these principles in mind.

Part 4- Proof of Concept

This section provides a proof of concept for the profile by giving examples of the use of the profile for actual EAI modeling. An example is provided that is relevant to both of the scenarios of the Scope that are covered by this specification and uses collaboration modeling. In Chapter 7, a variant of part of this example is presented in the CCA of the UML Profile for EDOC.

Part 5 - Implementation Mappings

The profile presented in this specification is intended to provide the basis for modeling EAI architectures, largely at a logical level. However, the implementation of such an architecture requires, of course, the use of various technologies and tools appropriate to integration, such as message brokers. This section presents a selection of mappings of the modeling approaches of the profile into such implementation technologies. The set of technologies discussed here is by no means an exhaustive set of those applicable to EAI, but is simply intended to demonstration how the profile is usable with such technologies.

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:

- CBOP
- Charles Schwab & Co.
- Data Access Technologies
- DSTC
- Hitachi, Ltd.
- International Business Machines Corporation
- IONA
- Oracle Corporation
- Rational Software Corporation
- Unisys Corporation

Part 1 - Introduction

Contents

This section contains the following chapters.

Chapter	**Page**
1. Introduction and Guide	1-1
2. Scope	2-1
3. Modeling Approach	3-1
4. Compliance	4-1
5. Requirements and Areas for Discussion	5-1

Introduction and Guide 1

1.1 Introduction

As enterprises adapt to business change and new opportunities, they seek to build on their existing strengths and assets for competitive advantage. Electronic trading with consumers and other businesses is one of these trends. This frequently entails building new applications by coupling existing ones, which is known as Enterprise Application Integration (EAI). This is most often done with some form of messaging that provides loose coupling to make it easy to change, to link heterogeneous systems and operating environments, and to maximize resilience and robustness in cases of partial failure.

Enterprise Application Integration technology is being promoted to integrate legacy systems with new packages. But integrating legacy applications with new software is a difficult and expensive task due, in large part, to the necessity of customizing each connection that ties together two disparate applications. There is no single mechanism to describe how one application may allow itself to be invoked by another.

We intend to solve this problem by defining and publishing a metadata interchange standard for information about accessing application interfaces. The goal is to simplify application integration by standardizing application metadata for invoking and translating application information. Once these standards exist, tools may be constructed to facilitate the development, execution, and management of these integration points.

Such connected systems are inherently complex to define and manage. A well-known approach to managing complexity is to define levels of concern. Modeling with UML has been shown to be successful at representing differing levels of detail. The appropriate level for EAI is *application architecture* — the treatment of the interfaces and interactions between applications. UML has been used successfully for modeling at this level, and this specification presents the authors' view of best practice for using the existing UML for modeling application architectures, i.e., architectures composed by enterprises to enable application integration.

1.2 Attachments

XMI and DTD files for the EAI Metamodels can be found in OMG document number ad/2001-08-25.

Scope 2

The scope is described with three generic scenarios representing the evolution of the integration requirements:

- Scenario 1. Application integration through connectivity.
- Scenario 2. Application integration through information sharing.
- Scenario 3. Application integration through process collaboration.

For each scenario major characteristics and requirements are described. Obviously, scenario 2 requires the functionality described in scenario 1 and scenario 3 requires the functionality described in scenario 2. However, as we move forward from scenario 1 to scenario 2 and scenario 3, the underlying functionality becomes less visible and more and more hidden in the infrastructure.

As the industry moves forward, scenario 3-or an updated version of scenario 3-will most likely become the dominant scenario.

2.1 Scenario 1: Connectivity

A small set of applications has to communicate synchronously or asynchronously with each other to provide business functions.

It must be possible to model the following abstractions:

- Service requester and provider
- Synchronous and asynchronous service request
- Request, reply and notification

In this scenario, the participating applications share a common architecture. They share the data model of the communication and they are able to activate the appropriate applications to obtain a service.

There is a need for additional abstractions such as queues (local or remote) and topics. At one level, queues and topics should be invisible, but at a lower level of detail they may well be required.

2.2 Scenario 2: Information Sharing

This scenario comes from handling securities.

An investor orders a stock trade, typically by sending a message describing the stock trade to be carried out. (We discuss the creation of the message in the next scenario.) This stock trade order triggers a set of autonomous actions: checking the investor's account, checking the position of the institution, notifying a broker if the trade is large, and notifying a broker as well as the investor if there are any issues.

If the order is accepted the market place is selected and an institution such as a market maker executes the trade. After execution, the investor records are updated. Information about the executed trade is sent to the investor via pager and e-mail and to internal systems such as bookkeeping that require the information.

The securities firm is not only interested in handling requests properly but also in answering questions from investors, regulators and other interested parties, both internal and external, at any stage during or after a trade.

A key requirement is that it should be easy to add new participants and new functionality with no or minimal impact to existing participants and services.

A good way to deal with this scenario is to model it as information sharing between applications and actors, such as investors and brokers. Such information sharing can be implemented through publishing and subscribing to business events enabling communication between the participants. We assume that all applications reflect a shared understanding about the meaning and sequence of the individual business events and act according to this shared understanding. However, we will assume that applications and actors participating in these processes are isolated from knowledge about who will consume their information and in which topic and format the recipients expect it.

It must be possible to model the following abstractions:

- Messages representing business events. (We are much less interested in messages that do not represent business events.)
- Publication of messages and business events-the ability to share information.
- Queues and topics-it must be possible to separate output containers of sending applications from input containers of receiving applications.
- Data transformation-each program must be able to create or consume messages in its own format. Applications should be able to use data structures suitable to their own language, e.g., a C++ program should not have to handle SWIFT or XML formats. Data transformation has to include data verification.
- Propagation-the ability to use any protocol to receive or deliver a message, including the allocation of a received message to a queue.

- Subscriptions to determine the receiving programs, their input containers or propagation routes, and their transformations. Subscriptions should be able to represent various cases, including interest of users, data routing, activation of programs.
- Retention to keep the history of relevant messages from creation through stages of processing, transformation, and consumption.
- Auditing, tracking, and mining-the ability to find and relate messages, both consumed and in flight.

In this scenario, the applications share a common business event and process model at the conceptual level. However, details of the layout of the data may vary, e.g., one program may use SWIFT structures, while another uses XML.

The term information sharing is used to characterize the interaction between participants providing information for the right recipients. Where time is of the essence and information is communicated with messaging/event technology we refer to zero latency information sharing.

2.3 Scenario 3: Process Collaboration

Company A offers its merchandise through the Internet. While some customers order goods using a browser interface, the majority of the orders are communicated business-to-business (B2B) using one of the B2B protocols. In simplified form, a B2B protocol consists of the following business events:

- RFQ (Request For Quotation)
- Offer
- Acceptance
- Shipment notice
- Bill
- Payment

Other events involved in negotiations, inquiries, changes, cancellation, and other additional steps (e.g., steps involving communications problems) are not considered in this simplification.

Company A communicates with business partners over secure Internet channels. Non repudiation, high reliability (including disaster tolerance), exactly once semantics, fully automated user-accessible application-independent auditing and tracking are basic requirements. Outgoing communication will use the requested protocol. Messages representing business events are carefully checked for process, sequence and data accuracy. Any error will raise an exception condition. Incoming communication is checked carefully as well. Some errors may need manual correction, which needs careful documentation.

Company A offers the flexibility for customers to use their favorite B2B protocols as long as they can represent a proper order process.

Applications should be independent of the specifics of the business protocols, but it is assumed that the desired interaction with an application can be achieved using its interfaces. At least three levels of interface support can be distinguished in applications:

- Applications that are only able to react through activation of their interfaces.
- Applications that can accept requests and can notify the outside world using events. At least some of these applications have to be configured to activate the desired events.
- Applications that additionally provide a process interface. These applications have to be configured to use the desired process structure.

In any of these cases it can not be assumed that the process as seen by the application is the process as seen by the selected B2B protocol. Actually it is desirable to hide the internal processes from business partners, so they can be changed without impact to the outside world and potential competitive advantages can be hidden. To achieve this, a mediation service has to be available to transform the process and data semantics embedded in the B2B protocols to the process and data semantics of an enterprise's internal processes. This transformation will be called semantic mediation. Semantic mediation is part of the core functionality required for the integration of autonomous applications.

Flexibility in B2B communication requires a repository of information that governs communication with a particular trading partners. This information includes security (including application security), notifications, subscriptions, B2B protocols and their extensions and adaptations, and indications for internal routing. It should be possible to group trading partners according to various criteria. This information comprises what is often termed a trading community agreement.

To model this level of integration it must be possible to model the following abstractions in addition to the abstractions defined in the previous scenarios:

- Semantic mediation-the ability to transform process and data structures between applications and B2B protocols.
- Propagation between enterprises-secure, with non repudiation, exactly once semantics, and disaster protection.
- B2B-level auditing, tracking, and mining-a business event can be reviewed, analyzed in its process context, and mined for insight into business behavior.
- B2B protocols-processes based on the communication of business events or business events in the context of process and customer relations.
- Trading community agreements in a repository containing information about trading partner and the communications with them.

This specification addresses primarily the first two scenarios. It provides enablement for scenario 3, but this scenario requires other elements that go beyond its scope. Scenario 3 is included here to clarify the relationship to work going on in ebXML and elsewhere.

Modeling Approach 3

The EAI specification is delivered as a complete MOF-based metamodel and a UML profile, which actually consists of two profile, one for collaboration modeling and one for activity modeling. This approach facilitates exchange with both UML tools and MOF-based tools/repositories.

3.1 Metamodel

As is the common practice, the MOF-based metamodel is captured as an object-oriented model expressed using a suitably restricted subset of the UML notation. The UML elements used in this specification are:

- Classes with attributes and (query) operations.
- Binary associations, where composite and navigation adornments are permitted. Association classes and qualified associations are not permitted.
- Packages, including nesting and imports.
- The object constraint language, OCL, for expressing well-formedness constraints.

The EAI metamodel is documented using the following conventions:

- The overall structure of the metamodel is shown as one or more package diagrams, depending on the level of nesting required.
- At the lowest level, packages are limited in size, and only one class diagram per package is required.
- In explaining a package, the important collaborations between classes are identified and described as one. Individual classes are described separately where this enhances the overall understanding of the model.
- Well-formedness constraints are also grouped with the collaborations to which they are relevant.
- The semantics of each collaboration is described as specified below.

3.2 UML Profile

The UML profile allows modelers to use UML as a concrete notation for producing EAI models using UML modeling tools which support the UML extensions mechanisms, chiefly stereotypes, tagged values and custom icons. Some tools are available, e.g., [ref objecteering], which can accept a profile definition and configure a modeling tool to force modelers to conform to that profile by using only elements of the UML subset and only the stereotypes, tagged values and icons declared in the profile.

A mapping between the metamodel and the UML profile is defined as part of the EAI specification. This is intended as a basis for the development of tools that will transform models expressed using the UML profile into models conforming to the metamodel, and vice versa. The details of the mapping are given as part of the definition of the profile.

3.3 Four-layered Architecture

The relationship of the EAI specification to the four-layered architecture defined by the OMG is as follows. MOF is at level 3, so the EAI metamodel is at level 2. The EAI UML profile is also at this level - it is just a set of additional constraints (what stereotypes, tagged values, etc.) on how UML is to be used when notating EAI models. The EAI metamodel should be thought of as the definition of the abstract syntax of EAI models. An EAI model, which is at level 1, is an expression of this abstract syntax. An EAI model is a specification of the architecture of an event-based system and the allowable information flows through that system. Level 0, then, represents actual behaviors of an event-based system, for example a particular instantiation of the architecture or a particular message flow through that system. These behaviors and instantiations must conform to the specification of behavior captured by the EAI model.

3.4 Semantics

There are a number of approaches to semantics. One is to describe how a model (in this case an EAI model) constrains the set of possible behaviors at M0 which satisfy that model. This can be captured formally by explicitly modeling (in some formal language) the structure of the abstract syntax, the structure of M0 behaviors and the relationship between the two. However, a formal definition can be somewhat inaccessible. The approach taken in this specification is to describe the semantics in English, using a model of M0 behaviors to help clarify the explanation where appropriate.

Compliance 4

4.1 Overview

Compliance with this standard by a vendor can be partial. To facilitate this the compliance points have been defined separately (Section 4.2, “Compliance with the UML Collaboration Profile,” on page 4-1, Section 4.3, “Compliance with the UML Activity Profile,” on page 4-2, and Section 4.4, “Compliance with the MOF-based EAI Metamodel,” on page 4-2) and examples of plausible compliance statements are provided (Section 4.5, “Compliance Statement Examples,” on page 4-3).

References to other OMG standards are abbreviated in the compliance point definitions, but in all cases refer to the specific revisions listed in the table below:

Standard	Version Referenced
UML	1.4
XMI	1.2
MOF	1.3

4.2 Compliance with the UML Collaboration Profile

The UML Collaboration Profile for EAI is defined in Chapter 8.

4.2.1 General Compliance

A compliant implementation supports the UML XMI exchange mechanism for the UML packages extended by the Collaboration Profile for EAI. It also supports the UML exchange mechanism for the stereotypes and tagged values defined by the Profile.

The UML packages that the Profile extends are "Behavioral Elements::Collaborations" plus the transitive closure of all of the packages upon which that package depends.

An implementation that satisfies the General Compliance point can be described as one that "complies with the UML Collaboration Profile for EAI."

4.2.2 Visualization

An implementation satisfies the visualization compliance points if it supports the UML notation for the packages extended by the Collaboration Profile and for the EAI extensions to those packages. An implementation that complies with the Collaboration Profile may or may not satisfy the Visualization compliance point.

An implementation that complies with the Collaboration Profile and that satisfies the Visualization compliance point for the Profile can be described as one that "complies with the UML Collaboration Profile for EAI including UML notation."

4.3 Compliance with the UML Activity Profile

The UML Activity Profile for EAI is defined in the Activity Modeling chapter.

4.3.1 General Compliance

A compliant implementation supports the UML XMI exchange mechanism for the UML packages extended by the Activity Profile for EAI. It also supports the UML XMI exchange mechanism for the stereotypes and tagged values defined by the Profile.

The UML packages that the Profile extends are "Behavioral Elements::Activity Graphs" plus the transitive closure of all of the packages upon which that package depends.

An implementation that satisfies the General Compliance point can be described as one that "complies with the UML Activity Profile for EAI."

4.3.2 Visualization

An implementation satisfies the visualization compliance points if it supports the UML notation for the packages extended by the Activity Profile and for the EAI extensions to those packages. An implementation that complies with Activity Profile may or may not satisfy the Visualization compliance point.

An implementation that complies with the Activity Profile and that satisfies the Visualization compliance point for the Profile can be described as one that "complies with the UML Activity Profile for EAI including UML notation."

4.4 Compliance with the MOF-based EAI Metamodel

There is a separate and independent compliance point for each of the MOF metamodels defined in this specification.

A compliant implementation of a metamodel supports exchange based on the XMI DTD generated from the metamodel.

The metamodels defined by the specification and the corresponding generated XMI DTDs are as follows:

EAI MOF based metamodel	Chapter in which the metamodel is defined	XMI DTD
Integration	6	CM4EAI
TDLang	7	TDLang
TypeDescriptor	7	TypeDescriptorTDLang
COBOL	14	COBOLtdlang
PL/I	14	pliTDLang
C	14	tdlang
C++	14	cpptdlang

The language metamodels depend on the TDLang and typedescriptorTDLang XMI DTDs.

There are no specific requirements for visualization of the EAI Metamodel.

A compliant implementation of the Integration metamodel can be described as one that "complies with the EAI Integration metamodel;" a compliant implementation of the COBOL metamodel can be described as one that "complies with the EAI COBOL metamodel;" etc.

4.5 Compliance Statement Examples

Any combination of the compliance points can be used. Examples of compliance statements follow:

- Tool XXX complies with the UML Collaboration Profile for EAI.
- Tool XXX complies with the UML Collaboration Profile for EAI including UML notation.
- Tool XXX complies with the UML Activity Profile for EAI.
- Tool XXX complies with the UML Activity Profile for EAI including UML notation.
- Tool XXX complies with the UML Collaboration and Activity Profiles.
- Tool XXX complies with the UML Activity Profile including UML notation and with the UML Collaboration Profile.
- Tool XXX complies with the UML Collaboration Profile including notation and with the UML Activity Profile including notation.
- Tool XXX complies with the UML Collaboration and Activity Profiles, including UML notation for both. (Note: this statement is equivalent to the previous one.)

- Tool XXX complies with the EAI C Metamodel.
- Tool XXX complies with the EAI C++ Metamodel .
- Tool XXX complies with the EAI Integration, C, C++, and PL/1 metamodels.
- Tool XXX complies with the UML Collaboration and Activity Profiles including notation for both. It also complies with the EAI Integration, COBOL, and PL/1 metamodels.

Requirements and Areas for Discussion 5

5.1 Mandatory Requirements

The RFP requirements are quoted in italics, followed by a summary of the specification's response.

Responses shall propose a UML profile suitable for modeling at the architectural level, as distinct from business modeling or application system design. The purpose is to represent IT systems (existing systems, vendor-supplied packages and newly developed application systems) at the level appropriate for integration between them.

This UML profile is suitable for modeling at the architectural level because it is based on a metamodel that supports composition and decomposition, and it defines interfaces to systems, applications and packages, both new and existing.

5.1.1 Event-Based Architecture

Proposals shall provide the means for specifying architectures and processes based on the flow of business events. They shall provide for:

- Integration of applications based on the occurrence of events.
- Description of a process or information flow as a series of business events.
- Architectural decomposition of the defined processes or information flows into their implementing applications, each the sender or receiver of events.

The principal means used for showing event-based flows is in UML collaboration diagrams (*Collaboration Modeling* chapter 8). An alternative representation using activity graphs is given (*Activity Modeling* chapter 9). The modeling elements defined are well known in the industry as popular means for constructing event-based flows. The Integration metamodel (*EAI Integration Metamodel* chapter 6) inherits the composition mechanism of the Flow Composition Model (FCM) in the UML Profile for EDOC, which models senders and receivers as sources and sinks.

5.1.1.1 Modeling Elements

They shall provide for loose coupling including, but not necessarily restricted to, the use of messaging and message brokers. At least the following elements shall be included:

- Publish and Subscribe - distribution of messages based on dynamically varying subscriptions
- Routing, fanout and filtering of messages - distribution based on rules or attributes
- Validation and transformation (mapping) of messages
- Augmentation (enrichment) and correlation (fan-in) - adding data (e.g., from a database) and accumulating data from related messages
- Deadline and post-dating - checking on-time arrival and avoiding early delivery
- Exception processing

All these elements are defined in the metamodel and mapped into the profile. In addition, the basic semantics of messaging and queueing are defined, as this paradigm is commonly used for loosely coupling systems. The use of message brokers is demonstrated in the non-normative mapping to a commercial product (*Mapping to WebSphere MQ Integrator* chapter).

5.1.2 Heterogeneous Environment

Proposals shall be generally applicable to heterogeneous networks of programming systems, operating systems, application systems, servers and packages. They shall show how they apply to mixtures that include ORBs, Internet servers and other subsystems.

The message-based approach adopted in this specification is well known to be applicable across a wide variety of systems and servers. Heterogeneity of programming systems is shown in the normative language models (Chapter 14). Interfaces to several subsystems are given in Appendix A. The sample in *Example: Connectivity and Information Sharing* chapter illustrates the use of several servers and protocols (both networking and data interchange protocols) across more than one enterprise.

5.1.3 XML

Proposals shall show how they are compatible with the use of XML in message formats for business data.

The model allows messages to be in different formats, including XML. The message metamodel includes a format specification that acknowledges the distinction between self-defining and separately defined message formats (see Section 6.3.4, “EAIMessageContent,” on page 6-7). In the case of XML, generic DTDs or schemas can be defined as a domain, or a message format may comply with a specific DTD in another domain.

5.1.4 XMI

Proposals shall show how they are compatible with the rendering of metadata in XMI. Models shall be interchangeable between different tools through the use of XMI. This shall include the use of XMI to define message formats and the CWM metamodel for transformations.

Because we have not extended the OMG MOF, XMI can be used to interchange EAI models. XMI files for the EAI metamodels are provided as supplementary material to this specification.

The EAI transformation operator neither prescribes nor proscribes the way in which transformations can be defined. Rather, the metamodel and profile furnish the means by which interfaces may be modeled. The architect or designer can simply name the salient attributes of messages or define interfaces in greater detail using the EAI Common Application Metamodel (CAM, see Chapter 7). The transformation details are left to the implementation, and this includes the case where a transformation tool is based on XMI and the CWM, which is an alternative to the use of CAM with different representations.

5.1.5 UML Profile for EDOC

Proposals shall show how they are aligned consistently with the UML Profile for EDOC.

The joint specification to the EDOC RFP includes a model for composing flow components (see OMG document ad/01-06-09, UML Profile for Enterprise Distributed Object Computing). This Flow Composition Model (FCM) unifies component composition and coordination both for events and for other styles of communication. This model is at a higher level of abstraction than the message flows, sources, targets, adapters, and operators in EAI. The model is applicable to EAI, and the EAI Integration metamodel (see Chapter 6) is a specialization of it. In particular, we use it to represent:

- Simple and compound flow components
- Input and output data sets
- Data flows, control flows and guard conditions

The EAI Integration metamodel includes publish/subscribe elements, and these are related to the Events Profile in EDOC as follows:

EAI Integration metamodel	**EDOC Events Profile metamodel**
EAISubscription	Subscription
EAISubscriptionFilter	NotificationRule
EAIMessage (in context of PublicationOperator and SubscriptionOperator)	PubSubNotice

The UML Profile for EDOC defines a Component Collaboration Architecture (CCA), part of the Enterprise Collaboration Architecture (ECA). Section 6.5, "CCA Component Library for EAI," on page 6-39 presents a mapping between the CCA and the EAI Integration metamodel.

In the EAI CAM (Chapter 7) the links are as follows:

- Both the Java language metamodel and the FCM metamodel in EDOC require CAM to tie parameters into the data typing and type composition structure that the metamodel provides.
- The Java metamodel has associations to the TDLang metamodel. Java typed elements implement the TDLangElement class, while the associated simple and complex Java data types inherit from TDLangClassifier class. Java classes will implement the TDLangComposedType.
- The FCM metamodel contains FCMParameter class, which associates to TDLangElement class. FCMParameter represents a data bytestring of the elements. Contents of the bytestring are mapped to the associated element by TDLangElement.

5.1.6 MOF Alignment

Proposals shall conform to the OMG MOF.

No extensions to the OMG MOF are proposed. The EAI Integration metamodel and the EAI Common Application Metamodel are based on MOF

5.1.7 Proof of Concept of Profile

Submissions shall provide sample models expressed in terms of the profile.

Part 4 shows sample models that use class diagrams and collaboration diagrams of the profile. These come from from the financial services industry. Chapter 9 shows an example of activity graph usage with the profile. The presentation in behavioral diagrams will be considered non-normative.

5.1.8 Demonstration that Models are Implementable

Submissions shall show that models built using the profile will map to practical implementations using generally available products and services. An acceptable example of such a demonstration for the publish/subscribe elements of the profile would be consistency with one or more of the CORBA Notification Service, JMS or OAMAS. For other elements, an example would be a mapping to a commercially available class of products, e.g., message brokers or mail routers.

A non-normative mapping to an implementation in JMS is given in Chapter 13 - "Java Message Service (JMS)". This demonstrates not only the publish/subscribe elements but also those that support direct messaging. A non-normative mapping to IBM's WebSphere MQ (formerly MQSeries) and WebSphere MQ Integrator (formerly

MQSeries Integrator) is given in Chapter 12 - “Mapping to WebSphere MQ Integrator”, which is, in fact, a specialization of the metamodel on which the UML Profile for EAI is based.

5.2 Discussion issues

5.2.1 Development and Management Aid

Submissions should discuss how the submitted profile aids or simplifies the architecture, development and management of EAI systems and solutions.

The profile defines the principal modeling elements needed for an IT architect or designer to take a business-level model or view and create an event-based architecture for EAI. Tools can be built to enable a designer take such a model to the next level of refinement (see below). With suitable instrumentation of run-time infrastructure, middleware and applications, monitoring and reporting of the behavior of executing systems is possible in a way that highlights bottlenecks, inconsistencies and other management problems by relating and comparing to the original or revised models. Such a feedback loop enables continuous process improvement.

5.2.2 Tool Support

Submissions should discuss how the submitted profile enables tool support for EAI definition and management and how such tools can be judged more or less compatible with the profile.

Tool compliance is discussed in Chapter 4. By providing a metamodel as the underpinning to the profile, the specification enables UML-based modeling and design tools to be coupled with implementation and configuration tools. A model developed with this profile can be converted to a high-level implementation model. For example, as a result of the mapping from the Integration metamodel to WebSphere MQ Integrator (WMQI) in Chapter 12, a UML tool can be written that exports an architectural model defined with this profile to an outline model in WMQI. A designer could then use the WMQI tool to complete the lower-level and implementation details of the model prior to testing and deployment.

5.3 Relationship to Envisioned OMG Technology

This section describes the relationship, in terms of alignment, reuse or overlap with OMG standards for which RFPs have been issued but which have not yet been adopted.

5.3.1 Real-time

The UML Profile for Scheduling, Performance and Time (from the Real-time PSIG, OMG document number ad/01-06-14) emphasizes the definition of quality of service (QoS). The UML Profile for EAI makes provision for QoS specifications in the

provision of streams (Section 6.4.1.2, “EAIStream,” on page 6-23) and resources (Section 6.3.7, “EAIQueue,” on page 6-12). These are left non-specific in this specification and can be augmented with specifications from the UML Profile for Scheduling, Performance, and Time.

5.4 Relationship to Existing Standards

5.4.1 UML

As a UML profile, this specification defines uses of UML 1.4 for the purposes of application integration. This includes classes and stereotypes.

5.4.2 Meta Object Facility (MOF)

UML is MOF compliant. This specification defines UML elements and adds additional semantics appropriate to the context of event-based architectures in EAI. Chapter 2 presents a metamodel in which each class is a MOF Class instance at the M2 level.

5.4.3 Common Warehouse Metamodel (CWM)

The Common Warehouse Metamodel (CWM) defines and publishes a metadata interchange standard for data warehousing and business intelligence tools and resources.

CWM gives metamodels for generic data structures that include XML documents, COBOL records, C structures and SQL schemas. These are aimed at data stores but are generic. They could be applied to message content descriptions. This level of refinement is a natural progression from the architectural designs supported by the UML Profile for Event-based Architectures in EAI.

CWM is highly reusable and is independent of any particular tool or data resource. It reduces the work required to integrate data warehousing and business intelligence tools.

CWM is needed for data transformation in a data warehousing and business intelligence environment. It provides data type mapping between a mix of different data resources, facilitates data translations from one data resource into another, allows data driven impact analysis for data lineage and allows data resource schemas to be viewed by developers.

The EAI Common Application Metamodel (CAM), which is described in Chapter 7, defines and publishes a metadata interchange standard for information about accessing enterprise applications such as CICS and IMS. CAM is reusable and is independent of any particular tool or middleware. It is likely to provide an incentive to connector suppliers by reducing the work required to create and develop connectors and/or connector builder tools.

CAM is needed for data transformation in an enterprise application integration environment. It provides data type mapping between mixed languages and facilitates data translations from one language and platform domain into another, it will allow data driven impact analysis for application productivity and quality assurance, and it will allow programming language data declarations to be viewed by developers.

In CAM a language metamodel, such as the COBOL metamodel, is used by enterprise application programs to define data structures which represent connector interfaces. It is important for connector tools to show a connector developer the source language, the target language and the mapping between the two. The CAM language metamodel also includes the declaration text in the model. This permits the connector/adapter developer to see the entire COBOL data declaration, including comments and any other documentation that would help him/her understand the business role played by each field in the declaration.

While CWM focus on data resources, CAM is for applications. CWM and CAM complement each other; both are needed in an enterprise IT environment.

5.5 Other Related Activities

The RFP states that submissions may deal with business-to-business (B-to-B) models as well as intra-enterprise models. However, there are other significant standards activities in B-to-B, and this specification does not address the area directly. EAI is a valuable underpinning to B-to-B along with other facets such as process modeling, which is addressed to a certain extent in the UML Profile for EDOC Business Processes Profile. To offer public services and interfaces to trading partners, an enterprise has to ensure that it has well-defined interfaces and well-architected systems. Much trading is inherently event based, and so streams, messages, publications, sources, targets, filters, transformations and other operations are natural modeling elements for the intra-enterprise systems that are needed to support both internal and public electronic trading. Hence, EAI is important both to inter and intra-enterprise business processes.

B-to-B modeling is dealt with in ebXML, which is based on a particular approach to B-to-B implementation. However, there are other approaches, including web services (SOAP, WSDL, UDDI and BPEL4WS) at W3C and OASIS, RosettaNet, OBI, EDI, OAG BODs and several industry-specific formats and protocols. BPML is a rival to BPEL4WS, which can be used to specify workflow and intra-enterprise processes as well as inter-enterprise processes. There continues to be a high volume of activity and a rapid rate of change.

Part 2 - Metamodel

Contents

This section contains the following chapters.

Chapter	**Page**
6. EAI Integration Metamodel	6-1
7. EAI Common Application Metamodel	7-1

This section describes the EAI metamodel, which, as explained in Chapter 6, is MOF compliant. The metamodel captures the essential EAI concepts. It may also be viewed as the abstract syntax of a language for specifying architectures for enterprise application integration. The metamodel is in two sections:

- The Integration Metamodel dealing with connectivity, composition and behavior
- The Common Application Metamodel dealing with interfaces and formats

Chapter 7 describes a UML profile for the language of the Integration Metamodel. It defines how UML (and therefore UML modeling tools) can be used as a concrete notation for this language.

EAI Integration Metamodel 6

6.1 EAI Integration Specializes FCM

The EAI Integration metamodel is a specialization of the Flow Composition Model from the UML Profile for EDOC (OMG Document Number: ad/2001-06-09, Part I, Chapter 5, Section 2). The following sections make extensive use of terms described in the FCM, and consequently it is assumed that the reader is familiar with it.

The UML Profile for EDOC also presents the Component Collaboration Architecture (CCA), part of the EDOC Enterprise Collaboration Architecture (ad/2001-06-09, Part I, Chapter 3, Section 2). In Section 6.5, “CCA Component Library for EAI,” on page 6-39 a mapping is presented between EAI Integration metamodel and the CCA. The mapping introduces the concept of a CCA "Component Library." Many of the concepts in EAI are represented as standard components that may be used in EAI compositions.

The EAI Integration metamodel reuses the concepts of flow, flow node and composition. It adds the following basic concepts which are required in EAI architectural modeling:

- Asynchronous communication
- Message queuing
- Message content and format

It additionally uses the FCM to define as flow components a number of concepts common to the message oriented middleware used in EAI, such a message routing, transformation, and publish/subscribe communication.

6.2 FCM support for recursive composition

The UML profile for EDOC provides support for the definition of ‘composite nodes’, whose function is defined by an FCMComposition. The FCMNodes in an FCMComposition may themselves be conposite. Terminals on a composite FCMNode

have an association with either an FCMSource or an FCMSink in the FCMComposition that defined an FCMCompositeNode. This is detailed in section x.x of the 'UML profile for EDOC' document number ad/xx-xx-xx.

6.3 EAI Specializations of the FCM

6.3.1 Motivation

Section 6.3 defines a set of specializations of the FCM. Each of these introduces a new concept required for EAI architectural modeling.

6.3.2 EAILink

Definition

Links between entities in an EAI architecture are often treated as event channels, and the occurrence of an event on such a channel initiates processing of the information associated with the event. As such, these links represent the flow of both data and control. In the FCM, data and control links are separate, so we introduce EAILink. EAILink inherits from FCMDataLink (which is a terminal to terminal link), and has an association with a single FCMControlLink.

Links may have their synchronization specified as synchronous, in which case a link between a pair of terminals implies a synchronous (call) invocation of the relevant FCMOperation, or asynchronous in which case a link between a pair of terminals implies an asynchronous invocation of the relevant FCMOperation (the FCMOperation which owns the parameter that the terminal represents).

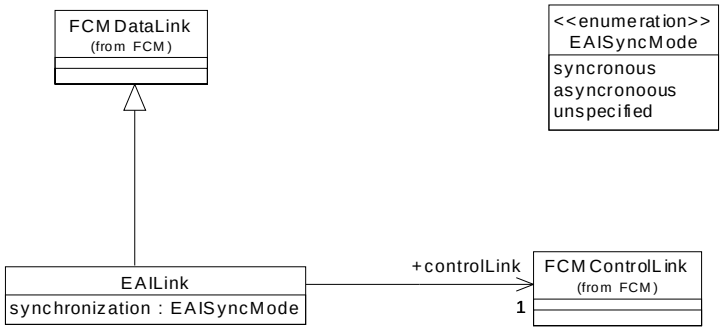


Figure 6-1 Definition of EAILink

Constraints

△

The source terminal of the EAILink is the same as the source terminal of its *controlLink*

context EAILink **inv**:

self.sourceTerminal = self.controlLink.targetTerminal

The target terminal of the EAILink is part of the *interface* of the *targetNode* of the controlLink

context EALink **inv**:

self.controlLink.targetNode.interface->exists(t | t=self.targetTerminal)

An EAILink connects two EAITerminals;

context EAILink **inv:**

inv: self.sourceTerminal.oclIsKindOf(EAITerminal)

inv: self.targetTerminal.oclIsKindOf(EAITerminal)

An EAILink connects two EAI Operators, sources or sinks

context EAILink

inv: self.sourceNode.oclIsKindOf(EAIOperator) **or**

self.sourceNode.oclIsKindOf(EAISource)

inv: self.targetNode.oclIsKindOf(EAIOperator) **or**

self.targetNode.oclIsKindOf(EAISink)

6.3.3 EAITerminal

△

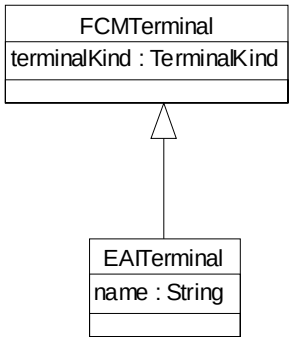


Figure 6-2 EAITerminal

Definition

An EAITerminal is a specialization of FCMTerminal.

Constraints

EAITerminal can be connected to other instances of terminals only via instances of EAILink.

(any link that can have a source terminal which is an EAITerminal must be an EAILink, any link that can have a target terminal which is an EAITerminal must be an EAILink)

context FCMComposition

inv: self.connections->forall(c | if c.oclIsTypeOf(FCMTerminalToNodeLink) then c.sourceTerminal.oclIsKindOf(EAITerminal) implies c.oclIsKindOf(EAILink))

inv:self.connections->forall(c | if c.oclIsTypeOf(FCMTerminalToTerminalLink) then c.targetTerminal.oclIsKindOf(EAITerminal)

An EAITerminal is the representation (see Figure 6-6) of an FCMParameter that is of type EAIMessageContent.

6.3.4 △*EAIMessageParameter*

Description

△An EAIMessageParameter defines the data to be processed by an EAIOperation. It is used to model an EAI message.

△An EAIMessageParameter conforms to a △message format specification, which may be physically manifest in the message (as, for example, with an inline XML DTD) or may need to be inferred by the MOM infrastructure. △In order to make this kind △of △distinction, △EAIMessageContent has two properties;

- **domain** which specifies the most generic message wireformat domain, and could be considered to encompass the domain of a generic parser. This is not restricted, but examples such as ‘XML’, ‘FixedFormat’, ‘Delimited’ would be valid
- **name** within the domain specified above, this is the name of the message format to be processed. This information is intended to allow message format handling infrastructure to identify what type of message within a particular domain is being processed, for example by reference to an XML Schema Document.

Subclasses of EAIMessageParameter can optionally be used to specify further structure in a message.

EAIMessageParameter inherits an association with TDLangElement, which may be used to specify details of the physical rendering of the message.

1. The TDLang metamodel provides an abstract view of the message element's structure. It may be used to represent both primitive and more complex data structures.
2. TDLang provides access to the language-specific representation of the message element (via the COBOL, PL/I, and other language metamodels in CAM), as well as its physical wire format (via the Type Descriptor Metamodel in CAM).

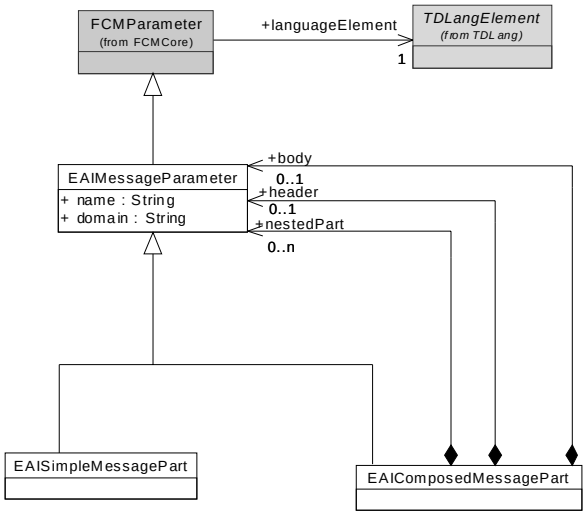


Figure 6-3 EAI MessageContent

6.3.4.1 EAISimpleMessagePart

This models a message part that does not contain other message parts. The format of a message element is defined in the MessageContent metamodel by its association to a TDLangElement, which is a class in the CAM (see Chapter 7). This link into the CAM provides all of the following for message elements:

6.3.4.2 EAIComposedMessagePart

Models a message or part of a message that is further composed of subparts. The model may optionally specify sub-parts that are considered to be ‘header’ or ‘body’ information.

6.3.4.3 EAIHeader

It is a common requirement for message processing to be able to specify a location to send any potential replies to, and to specify a location to which to send a message in the event of a message processing error. The information required to do this can be specified via a subclass of EAIHeader. In cases where the metadata contained in a header element does not concern replies or exceptions, it is not required for all headers in EAIMessageContent to be subclasses of EAIHeader,

EAIHeader is a subclass of EAIMessageElement; it has two associations to EAIMessageElement;

- replyTo: an EAIMessageElement that is required to specify the terminal to which replies to an instance of a message should be sent.
- exceptionTarget: an EAIMessageElement that is required to specify the terminal to which exception notices should be sent.

Constraint

The exceptionTarget and replyTo EAIMessageElement must not themselves be instances of the subclass EAIHeader

context EAIHeader

inv: replyTo->forall(rto | rto.oclIsKindOf(EAIHeader) = false)

inv: exceptionTarget->forall(exc | exc.oclIsKindOf(EAIHeader) = false)

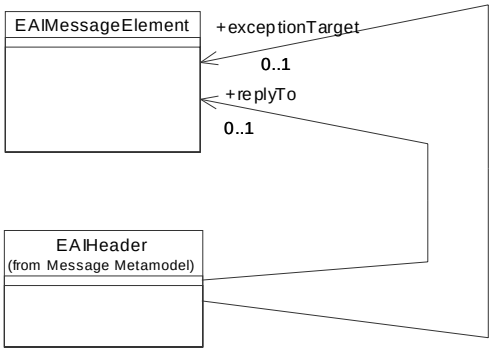


Figure 6-1 EAIHeader

6.3.4.4 EAIExceptionNotice

△Messages of this form may be sent △if △an exception occurs △during the processing of a message. An instance of an ExceptionNotice will normally contain the original message, with additional exception-specific information in a separate message part. In addition to these required message parts, the message may contain other message parts. These may be specified using the association to MessagePart inherited from EAIMessageContent.

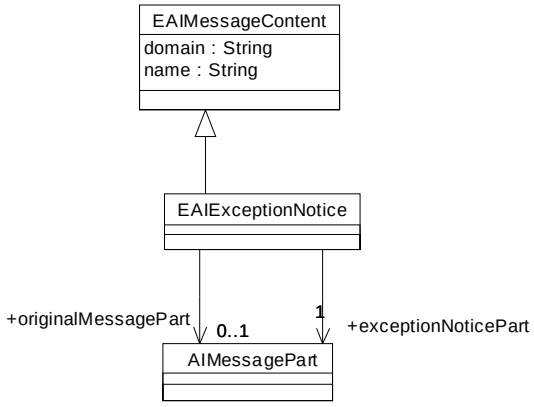


Figure 6-1 EAIExceptionNotice

6.3.4.5 EAIMessageContent

6.3.4.6 EAIMessagePart

EAIMessagePart may have two distinct elements:

1. A message header which contains metadata about the message rather than the application data itself. It is used to help determine processing either by middleware or by metadata-aware applications.
2. Message body, which contains the business content of the message.

The header and the body modeled via associations with EAIMessageElements.

6.3.4.7 EAIComposedMessagePart

EAIComoposedMessagePart is a subclass of EAIMessagePart which may itself contain messageparts.

6.3.5 EAIMessageOperation

Description

EAIMessageOperation is a subclass of FCMOperation used to describe operations for which all the inputs and outputs are messages.

Constraints

Every input and output of an EAIMessageOperation is an EAIParameter. EAIMessageOperation may have zero or one faults. If present, the fault must be and EAIParameter.

context EAIMessageOperation

inv: self.inputs->union(self.outputs)->forAll(oclIsType(EAIParameter))

inv: self.inputs->union(self.faults)->forAll(oclIsType(EAIParameter))

inv: self.inputs->size() <= 1

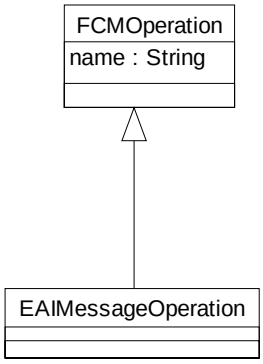


Figure 6-1 MessageOperation

6.3.6 EAISource and EAISink

Description

EAISource and EAISink represent points in an EAI architecture where messages appear (EAISource) and disappear (EAISink).

Sources and sinks may make use of EAIResources. An EAIResource represents a usable and sharable entity such as a queue (Section 6.3.7, “EAIQueue,” on page 6-9 or a database (Section 6.4.1.5, “EAIDBTransformer,” on page 6-27).

Constraints

EAISource is a subclass of FCMSource. Its sinks must be EAISink, and its implements operation must be an FCMOperation.

EAISink is a subclass of FCMSink. Its source must be an EAISource.

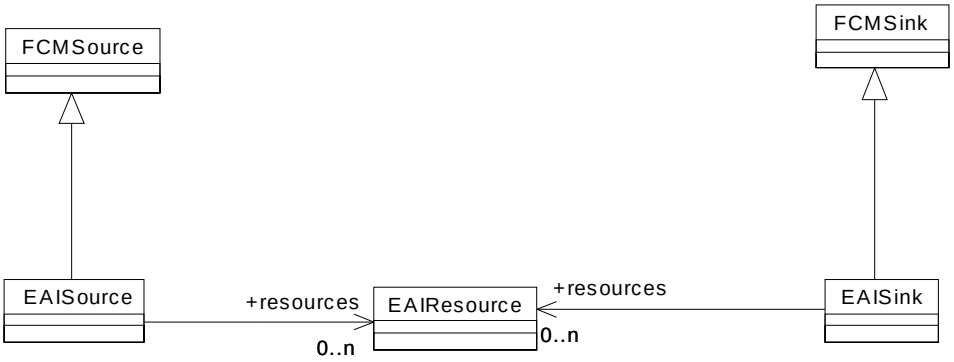


Figure 6-2 Sources and sinks

6.3.7 EAIQueue

Description

EAIQueue is a queue of finite or unbounded length, and is modeled as a subclass of EAIResource.

EAIQueue has a name and a Boolean “isBound” showing if the queue length is finite or unbounded. EAIQueue also has a maxLength, which specifies the maximum number of messages it can hold.

EAIQueue is restricted to holding a specific type of message content if an EAIMessageContent is specified for EAIQueue. Otherwise, EAIQueue can hold any type of message content.

EAIQueue is intended to be an abstraction of queuing infrastructure. We note that most MOM implementations allow machine-to-machine communication via a remote queuing infrastructure that can specify a number of different queue types and relationships between then. This can be modeled as refinement or realization of EAIQueue or (see Section 6.4.1.2, “EAIStream,” on page 6-24) of the EAIPrimitiveOperator EAIStream.

△

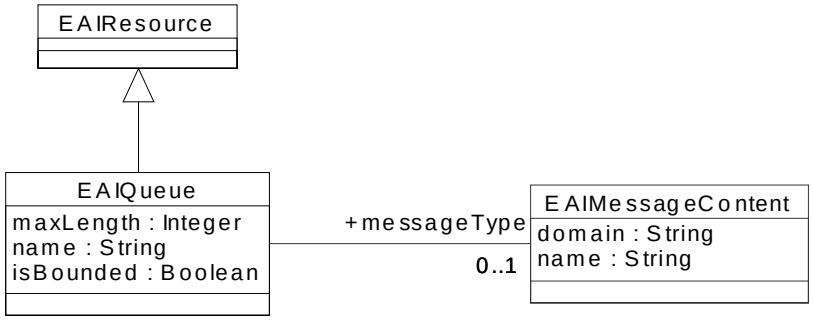


Figure 6-3 EAIQueue

Constraints

maxLength >= messages->size()

6.3.8 EAIQueuedInputTerminal and EAIQueuedOutputTerminal

A common means of implementing an asynchronous link between a pair of entities in EAI is for them to share a queuing infrastructure. In this case, the entity in which an event occurs places a message into a queue and then continues processing. The entity that is to act on this information can remove the message from the queue at any time. This normally involves the receiving entity doing one of the following:

1. Polling the queue for the arrival of a message
2. Blocking execution awaiting the arrival of a message
3. Being triggered by the arrival of a message

EAIQueuedInputTerminal and EAIQueuedOutputTerminal are subclasses of EAITerminal that are used to represent message communication tha occurs via queuing.

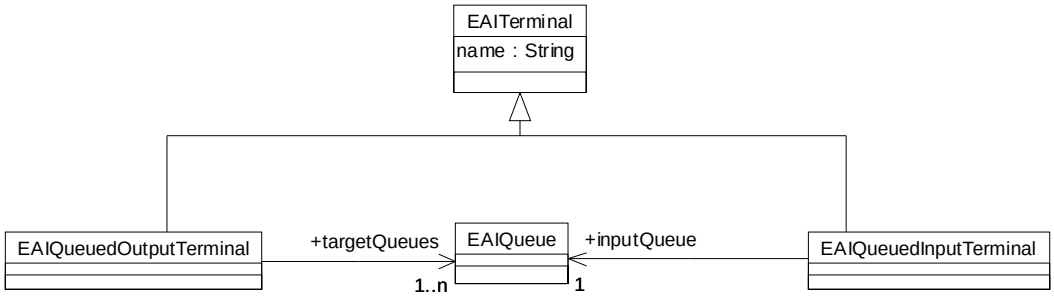


Figure 6-4 EAIQueuedOutputTerminal and EAIQueuedInputTerminal

An EAIQueuedInputTerminal has an association with the single queue that it reads from, while an EAIQueuedOutputTerminal has an association with each of the queues used by its target EAIQueuedInputTerminals.

Any operator that has an EAIQueuedOutputTerminal is understood to place a single copy of its output message on each of its targetQueues.

Queued input and output terminals may be used on any of the EAI constructs that have terminals (EAIPrimitiveOperator, EAICompoundOperator, EAISource, EAISink).

Constraints

All EAILinks from an EAIQueuedOutputTerminal as the sourceTerminal must have an EAIQueuedInputTerminal as the TargetTerminal.

The EAILink from an EAIQueuedOutputTerminal to an EAIQueuedInputTerminal must have synchronization=asynchronous.

An EAILink between an EAIQueuedOutputTerminal and an EAIQueuedInputTerminal implies that the inputQueue of the inputTerminal is in the targetQueues of the output terminal.

All EAIQueuedInputTerminals have EAILinks with all EAIQueuedOutputTerminals that use the same queue instance.

context EAILink

inv: if self.sourceTerminal.oclIsKindOf(EAIQueuedOutputTerminal) **then**

self.targetTerminal.oclIsKindOf(EAIQueuedInputTerminal) **and**

self.synchronization=asynchronous **and**

self.sourceTerminal.targetQueues->includes(self.targetTerminal.inputQueue)

6.3.9 EAIQueuedSource and EAIQueuedSink

Description

EAIQueuedSource and EAIQueuedSink are used to model the internal elements of an EAIMessageFlow (see Section 6.3.10.7, “Exposing terminals in an EAIMessageFlow,” on page 6-23) that is associated with EAIQueuedInputTerminals and EAIQueuedOutputTerminals.

When viewing the internals (i.e., the EAIMessageFlow) of a CompoundOperator, the element of the flow that receives messages (and passes them on to the rest of the flow) is a source of messages to the rest of the EAIMessageFlow, and vice versa. Hence, the part that reads from a queue is modeled as a EAIQueuedSource and the part that writes to a queue as EAIQueuedSink.

△

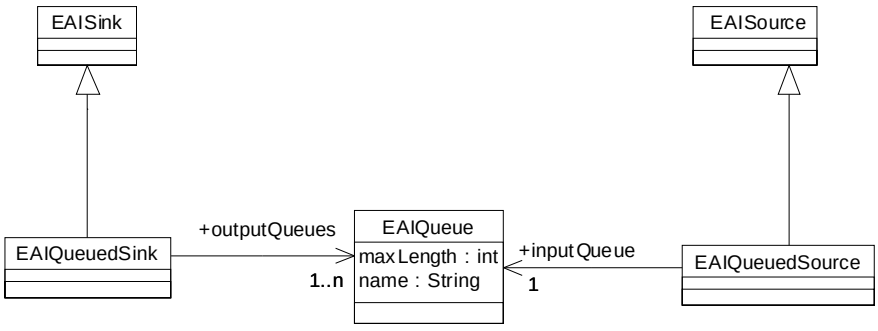


Figure 6-5 QueuedSource and QueuedTarget

Note that the terminals of EAIQueuedSink and EAIQueuedSource (used within the EAIMessageFlow) could themselves be queued terminals. This would imply that queueing is used both outside and inside the EAIMessageFlow.

Constraints

The outputQueues of an EAIQueuedSink must be the same as the targetQueues of the EAIQueuedOutputTerminal that it is associated with.

The inputQueue of an EAIQueuedSource must be the same as the inputQueue of the EAIQueuedInputTerminal that it is associated with.

Refinement relationships

An EAILink with synchronization of unspecified is refined by an EAILink with synchronization of either synchronous or asynchronous.

Where there is an instance of an EAILink with a synchronization of asynchronous linking a pair of FCMTerminals, this is refined by the substitution of EAIQueuedInputTerminal and EAIQueuedOutputTerminal for the FCMTerminals.

6.3.10 Operators

6.3.10.1 EAIOperator

Operators act upon messages as they flow between systems. We define △EAIOperator to be a subclass of FCMFunction△.

EAIOperators have a type, EAIType. An EAIOperator prototype can also be used to specify an EAIType. EAIOperator may optionally specify EAIResources that it uses to enact its function.

Constraints

context EAIOperator

Define what it means to be a prototype

let isPrototype = self.defines->size() = 1

let isInstance = self.defines->isEmpty()

An EAIOperator has the same number of terminals as its prototype;

inv: if isInstance then self.interface->size() = self.type.protoype.interface->size()

The prototype for a prototype is itself;

inv: if isPrototype then self.type.prototype = self

All of the terminals of an EAIOperator are EAITerminals;

inv: self.interface->forall(t | t.oclIsKindOf(EAITerminal))

An EAIOperator's terminals have the same names as its prototype;

inv: if isInstance then self.interface->

forall(t | self.type.prototype.interface->exists(tt| tt.name=t.name))

An EAIOperator has the same set of resources as its prototype;

inv: if isInstance then self.resources = self.type.prototype.resources

An

6.3.10.2 EAIResource

6.3.10.3 EAIType

EAIType is a subclass of FCMType. Its may have a single EAIOperation. An EAIType is defined by a prototype EAIOperator.

Constraints

context EAIType

EAIType has single operation

inv: self.operations->sizeOf() = 1

The single operation is the same as the ‘invokes’ operation of the prototype

inv: self.operations->any() = self.prototype.invokes

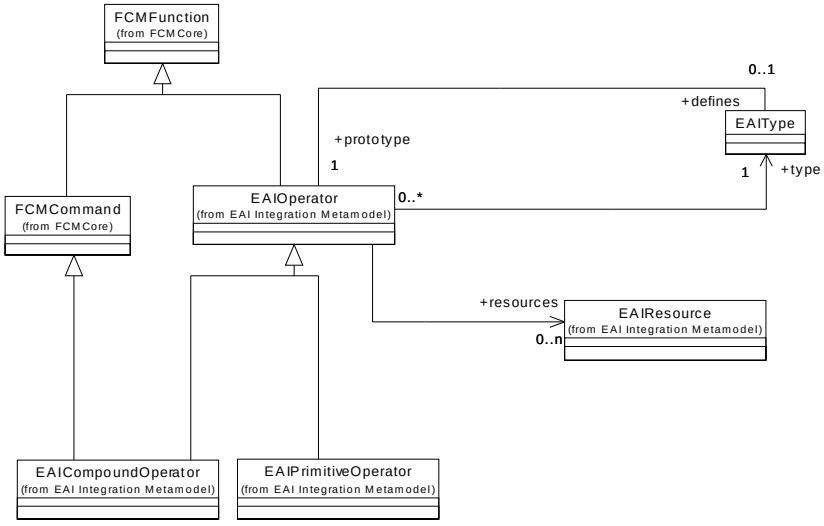


Figure 6-1 Definitions of PrimitiveOperator and CompoundOperator

6.3.10.4 EAIPrimitiveOperator

Description

Instances of EAIPrimitiveOperator enact a simple message processing operation. △EAIPrimitiveOperator is △a subclass of △EAIOperator

Constraints

Inherited from EAIOperator.

6.3.10.5 EAICompoundOperator

Description

An instance of an EAICompoundOperator composes more complex message processing behavior from EAIPrimitiveOperators, from other EAICompoundOperators or both. EAICompoundOperator is a subclass of FCMCommand. It achieves composition via an association (inherited from FCMCommand) with EAICompositeComponent, a subclass of FCMComponent.

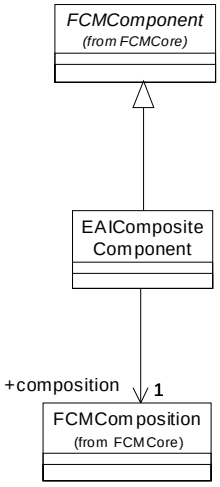


Figure 6-1 EAICompositeComponent

Constraints

context EAICompoundOperator

In addition to the constraints inherited from EAIOperator, EAICompoundOperator has the following;

EAICompoundOperator's function is *performedBy* a subclass of FCMComponent, EAICompositeComponent that has an association with FCMComposition;

inv: self.performedBy.oclIsKindOf(EAICompositeComponent)

The *performedBy* EAICompositeComponent for an instance is the same as its prototype;

inv: is isInstance self.type.prototype.performedBy = self.performedBy

Define the implementingComposition derived association

let implementingComposition = self.performedBy.composition

Define the *nodes* derived association

self.nodes = self.performedBy.composition.nodes

Define the FCMOperation implemented by the FCMComposition

let sourceOperation =

self.implementingComposition.nodes->

select(n | n.oclIsKindOf(EAISource))->any()->implements

The *invokes* operation (association inherited from FCMFunction) is the same as that implemented by the composition

inv: sourceOperation = self.invokes

inv: self.nodes = self.performedBy.composition.nodes

EAICompositeComponent

Description

EAICompositeComponent is a subclass of FCMComponent. It has an association with FCMComposition

Constraints

The FCMComposition associated with an EAICompositeComponent must contain an EAISource and one or more EAISinks.

context EAICompositeComponent

inv: self.composition.nodes->select(n | n.oclIsKindOf(EAISource))->sizeOf() = 1

inv: self.composition.nodes->select(n | n.oclIsKindOf(EAISink))->sizeOf() >= 1

It can only compose EAI-specific nodes;

self.composition.oclIsOfKind(EAIMessageFlow)

An EAICompoundOperator invokes an EAIMessageOperation.

6.3.10.6 EAIMessageFlow

An EAIMessageFlow is a subclass of FCMComposition. Each of its nodes (see Figure 6-2 on page 6-3) must be one of the operator classes (EAIPrimitiveOperator or EAICompoundOperator), and its connections must be EAILinks. In addition it allows nodes to have explanatory annotations attached to them.

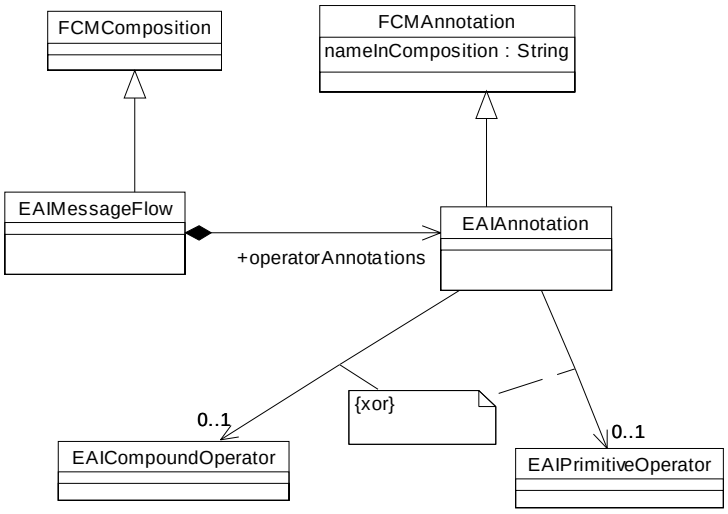


Figure 6-1 EAIMessageFlow

Constraints

context EAIMessageFlow **inv:**

self.nodes->forall(n : n.oclIsOfKind(EAIPrimitiveOperator) or n.oclIsOfKinf(EAICompoundOperator))

6.3.11 Adapters

An integration architecture provides paths for the flow of messages between the systems being integrated. Adapters provide the points at which the message-flow paths are actually connected to those systems. An adapter converts a specific kind of message from some system-specific format into a specified message-content type, or vice versa. △EAI adapters are modeled as a specialization of FCMFunction.

6.3.11.1 EAISourceAdapter

An EAISourceAdapter obtains information from a system, translates it into (some subclass of) EAIMessageContent and then sends it. Source adapters are modeled as a subclass of FCMFunction. The mapping between the internal format and the message is specified by an internalToMessage FCMMapping.

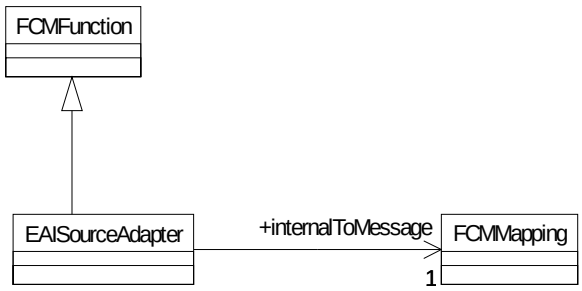


Figure 6-1 SourceAdapter

Constraints

An EAISourceAdapter has a single output terminal, which is an EAITerminal with the name “out”.

Output parameters of the invokes FCMOperation of SourceAdapter must be EAIParameters, which are associated with EAIMessageContent.

There is no constraint on the type of input terminals.

There is no constraint on the type of input and fault FCMParameters. It is noted that the faults FCMParameters may be EAIParameters (with EAIMessageContent) but that this is unlikely to be the case for input because adapters are used to link messaging to other (internal) interfaces.

6.3.11.2 EAITargetAdapter

An EAITargetAdapter has a single input EAITerminal ("in"). It receives a message with content of a given input type, maps the message content to the format required for a system and then delivers the information to the system. The transformation is specified by a messageToInternal FCMMapping.

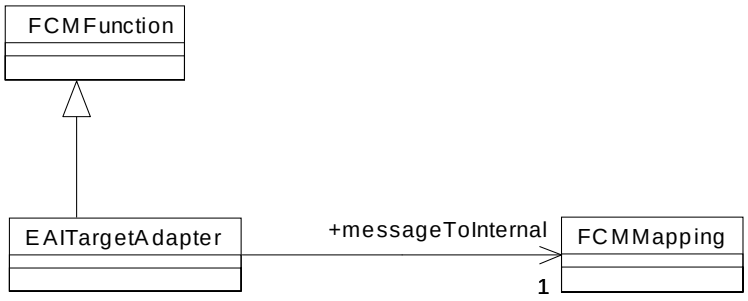


Figure 6-1 EAI Target Adapter

Constraints

An EAICallAdapter has two input terminals, one of which is an FCMTerminal that is *not* an EAITerminal and the other of which is an EAITerminal with the name “handleReply”.

An EAICallAdapter has two input terminals, one of which is an FCMTerminal that is *not* an eAITerminal and the other of which is an EAITErminal with the name “request”.

The input parameters of the FCMFunction that EAITargetAdapter invokes must be EAIParameters (with associated EAIMessageContent).

An EAITargetAdapter has a single input terminal, which is an EAITerminal with the name “in”.

There is no constraint on whether the outputs and faults of the invokes FCMFunction are FCMParameters or EAIParameters. However, they are unlikely to have associated EAIMessageContent because adapters are used to link messaging to other (internal) interfaces.

6.3.11.3 EAICallAdapter

An EAICallAdapter is invoked synchronously by an application that wishes to make use of a service made available via a server; the server accepts a request message and sends a response message back to the service requester. It has two input terminals:

- call": an FCMTerminal that a requesting application can use to invoke the call adapter
- handleReply": an EAITerminal that handles a reply

It has two output terminals:

- request": the EAITerminal from which the request message is sent
- out": an FCMTerminal to which the reply message is mapped

EAICallAdapter is a subclass of FCMFunction.

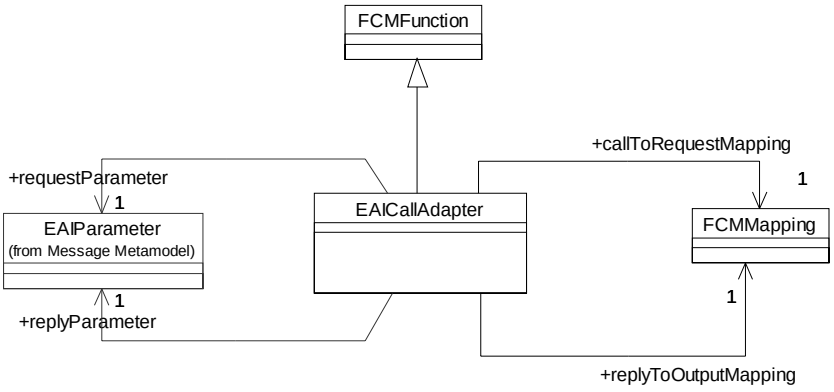


Figure 6-1 EAICallAdapter

The call adapter has two mappings, one of which specifies how the call input parameters are mapped to the request message; the other specifies how the return message is mapped to output parameters represented by the "out" terminal.

From the point of view of the requesting application, the EAICallAdapter is a single FCMFunction that takes an input on its “call” terminal and produces an output on its “out” terminal. Within the EAI model, this function is realized as follows (i.e., this is effectively the behavior of the FCMOperation invoked by the function).

1. Map the “call” input into a request message using the callToRequestMapping.
2. Place the request message on the “request” output terminal.
3. Wait for a reply message to be received on the “handleReply” input terminal.
4. Map the reply message to an output value using the replyToOutput mapping.
5. Place the output value on the “out” terminal.

Constraints

The parameter associated with the “out” terminal of an EAICallAdapter must be an EAIParameter with a message that is an EAIRequestReplyAdapter.

The FCMOperation invoked by an EAICallAdapter (when considered as an FCMFunction, see Figure 6-2) must have exactly one input FCMParameter and exactly one output FCMParameter.

The input FCMTerminal of an EAICallAdapter (that is *not* an EAITerminal) is associated with the input FCMParameter and the output FCMTerminal (that is *not* an EAITerminal) is associated with the output FCMParameter.

6.3.11.4 EAIRequestFormat

EAIRequestFormat is a subclass of EAIMessageContent that is used to specify a request message that may be produced by an EAICallAdapter and received by an EAIRequestReplyAdapter. While the structure of an EAIRequestFormat is just like any other EAIMessageContent, it a request message has the added semantic responsibility of identifying the terminal to which a reply to the message should be sent. How this identification is made is not explicitly defined in the metamodel syntax for an EAIRequestFormant, but it must be computable from the information specified for a request message (e.g., some sort of unique identifier for a reply terminal might be included in a header part of the message or some other sort of language element might be modeled to provided a logical identification of a terminal).

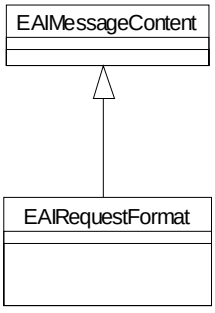


Figure 6-1 EAIRequestFormat

6.3.11.5 EAIRequestReplyAdapter

An EAIRequestReplyAdapter is used to synchronously invoke a function of a server application. It has two input terminals:

- "requestIn": an EAITerminal that accepts a message whose content is specified by an EAIRequestFormat (and thus provides some means of identifying a reply terminal)
- "handleReturn": an FCMTerminal that receives the reply from the server application

It has two output terminals:

- "replyOut": the EAITerminal from which the reply message is sent
- "call": an FCMTerminal to which the request is mapped to be sent to the server application

The request reply adapter has two mappings, one of which specifies how the "requestIn" input data are mapped to the server application call; the other specifies how the return data are mapped to the output message represented by the "replyOut" terminal.

From the point of view of the EAIModel, the EAIRequestReplyAdapter is a single FCMFunction that takes a request message on its “requestIn” terminal and produces a reply message on its “replyOut” terminal. This function is realized as follows (i.e., this is effectively the behavior of the FCMOperation invoked by the function).

1. Map the “requestIn” message into the data required for the server application call using the requestToCallMapping.
2. Place the call data on the “call” output terminal.
3. Wait for return data to be received on the “handleReturn” input terminal.
4. Map the return data to a reply message using the returnToReply mapping.
5. Place the reply message on the “replyOut” terminal and transmit it to the reply terminal identified in the request message.

Note that, in addition to simply being placed on the “replyOut” terminal, the reply message is transmitted to the reply terminal that is dynamically identified by the incoming request message. Request messages are generated by EAICallAdapters, with the reply terminal of the request message being the “handleReply” terminal of the EAUICallAdapter. Thus, the semantics of an EAIRequestReplyAdapter effectively results in the creation of a dynamic and temporary EAILink between the “replyOut” terminal of the EAIRequestReplyAdapter and the “handleReply” terminal of the EAICallAdapter that generated the request message.

Now, if the identified reply terminal is *not* an EAIQueuedInputTerminal, then the dynamic EAILink is considered to have *synchronization = unspecified.* The reply message is simply placed on the identified input terminal. However, if the identified reply terminal *is* an EAIQueuedInputTerminal (see Section 6.3.8), then the dynamic EAILink is considered to have *synchronization = asynchronous* and the reply message is placed on the inputQueue of the reply terminal.

△

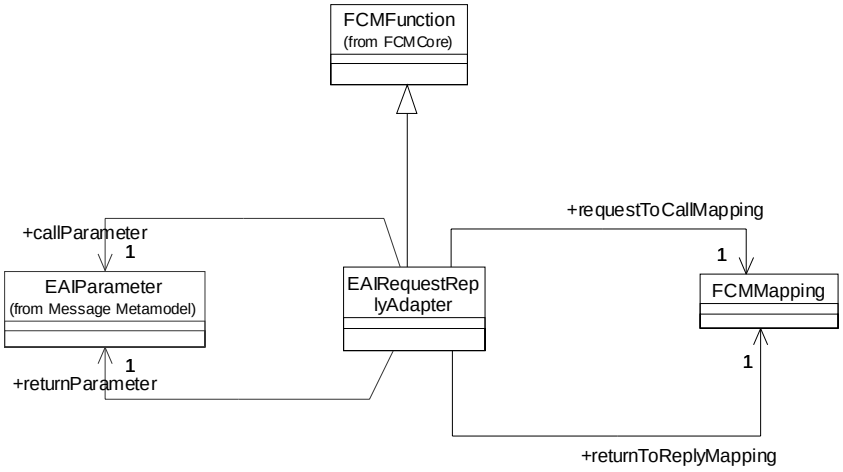


Figure 6-1 EAIRequestReplyAdapter

Constraints

An EAIRequestReplyAdapter has two input terminals, one of which is an FCMTerminal that is *not* an EAITerminal and the other of which is an EAITerminal with the name “requestIn”.

An EAIRequestReplyAdapter has two output terminals, one of which is an FCMTerminal that is *not* and EAITerminal and the other of which is an EAITerminal with the name “replyOut”.

The FCMOperation invoked by an EAIRequestReplyAdapter (when considered as an FCMFunction, see Figure 6-2) must be an EAIOperation with exactly one input EAIParameter, with a message that is an EAIRequestFormat, and exactly one output EAIParameter.

The “requestIn” terminal of an EAIRequestReplyAdapter is associated with the input EAIParameter and the “replyOut” terminal is associated with the output EAIParameter.

The representation of the callParameter of an EAIRequestReplyAdapter is the output FCMTerminal and the representation of the returnParameter of an EAIRequestReplyAdapter is the “return” terminal. (The representation association for an FCMParameter is shown on Figure 6-6 .)

6.4 Kinds of Operator

6.4.1 Operators

We define several specializations of EAIPrimitiveOperator and EAICompoundOperator. EAICompoundOperators combine more than one of the primitive EAI concepts represented by the PrimitiveOperators. Implementations of them do not need to follow this internal representation, provided that they obey the signature (in terms of the messages they receive and send) and the documented semantics.

6.4.1.1 EAIFilter

An EAIFilter is a subclass of EAIPrimitiveOperator.

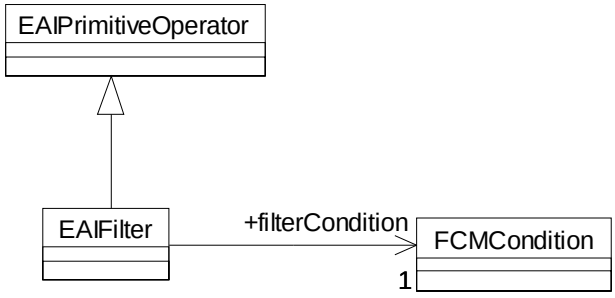


Figure 6-1 Filter

A filter has one input terminal and two output terminals. The output terminals must be named "true" and "false". If the message on the input terminal satisfies the filter condition, then it is copied to the output terminal named "true". Otherwise, the message is copied to the output terminal named "false".

6.4.1.2 EAIStream

EAIStream is an operator that allows 'quality of service' on a communication channel to be expressed.

The flow of control and data via EAILink between EAITerminals assumes that messages are always received in the order that they are sent and that there is basically no delay in their transmission.

In some implementations, a stream of messages may be received in a different order from that in which they are sent, and they may be received at a different rate from that at which they are sent. An EAIStream operator can be used to model this.

An EAIStream can be used to model reordering of incoming messages by maintaining a buffer.

In an implementation, an incoming message is may be added to the buffer in a place determined by the streaming algorithm. An outgoing message may be sent at the same or different time as an incoming message is received. The streaming algorithm determines when to place messages from the top of the buffer onto the "out" terminal. Typically, this will be when the buffer contains a sufficient block of messages in the correct order.

All of this behavior is abstracted via an emissionCondition that determines under what circumstances a message is emitted from the stream. The message emitted may be any element of the buffer. Once emitted from the stream, the message is removed from the buffer.

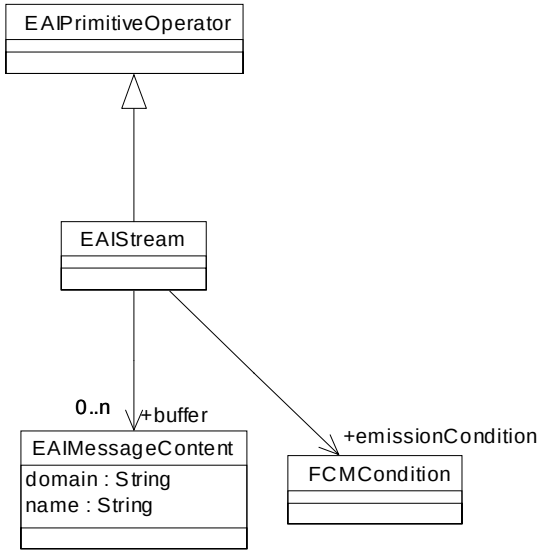


Figure 6-1 Stream

6.4.1.3 EAIPostDater

EAIPostDater is a subclass of EAIStream with a single input terminal ("in") and a single output terminal ("out").

On receipt of a message at its input terminal, it adds the message to the buffer, and creates an individual timingCondition for it. The timingCondition may entail a derivation from the content of the input message by a timerMapping. EAIPostDater holds the message until its individual timing condition is met, then emits it from its "out" terminal.

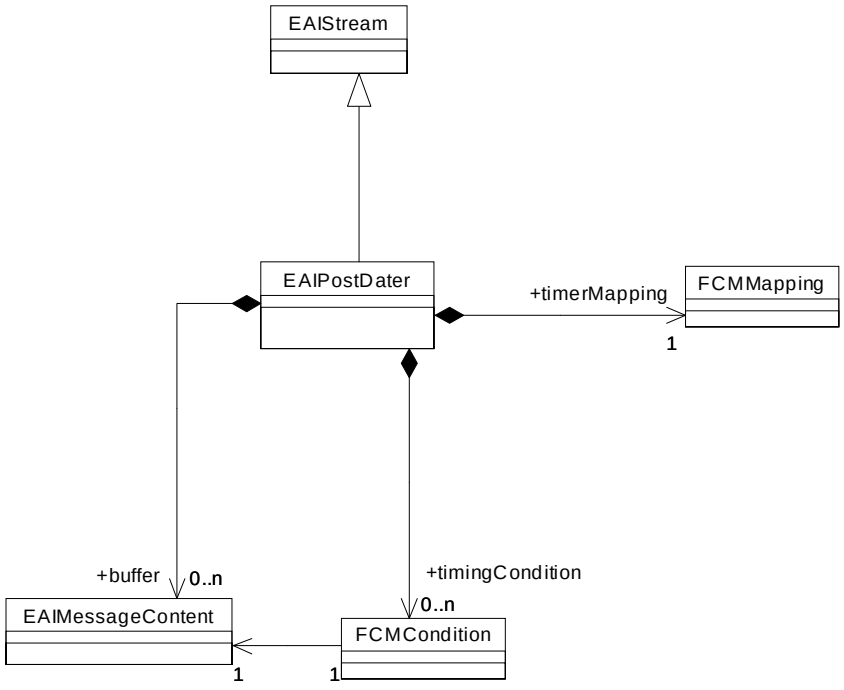


Figure 6-1 EAIPostDater

6.4.1.4 EAITransformer

A Transformer is a subclass of PrimitiveOperator with a single input terminal and a single output terminal.

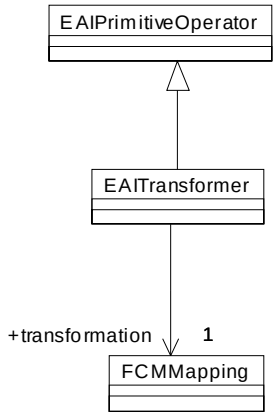


Figure 6-1 Transformer

The output message is a transformation of the input message, as dictated by the transformation FCMMapping.

6.4.1.5 EAIDBTransformer

An EAIDBTransformer is a subclass of EAITransformer that has access to an EAIDatabase.

EAIDatabase is modeled as a subclass of EAIResource and has the property databaseName. Subclasses of EAIDatabase may specify further properties such as information required to connect to the database.

An EAIDBTransformer is an EAITransformer, which is itself an EAIPrimitiveOperator, which may have resources attached to it (see Section 6.3.10.4). An EAIDBTransformer is specifically required to have exactly one such resource, which must be an EAIDatabase.

Access to a database as a resource allows the transformation to make use of information contained in the database. In particular, it allows the message to be augmented (or enriched) with data from the database.

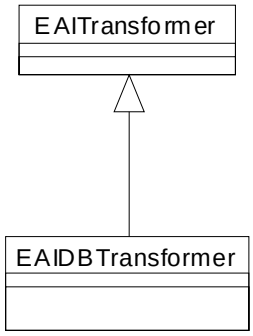


Figure 6-1 EAIDBTransformer

Constraints

An EAIDBTransformer has exactly one resource, which is an EAIDatabase.

6.4.1.6 EAIAggregator

An EAIAggregator is a subclass of PrimitiveOperator. It has a single input terminal ("in") and a single output terminal ("out"). Its purpose is to combine several messages (comprising an aggregate) into a single output message (EAIMessageAggregation). It is commonly used in conjunction with EAITimer, which can check for deadlines.

On receipt of a message, if there are no existing message aggregates, the aggregator creates one and adds the message to it.

On receipt of a subsequent message, the aggregator examines each existing aggregate, evaluating the addToAggregate condition (which will depend on the message header or body contents). If an aggregate exists for which addToAggregate evaluates to true, then the message is added to it.

Each time a message is added to an aggregate, the aggregateComplete condition is evaluated. If it evaluates to true, then a message is constructed from the messages it holds and is sent on the output terminal. The mapping from the messages contained in the aggregate to the message sent is specified by the aggregationMapping.

If the aggregateComplete condition does not evaluate to true, then no message is sent.

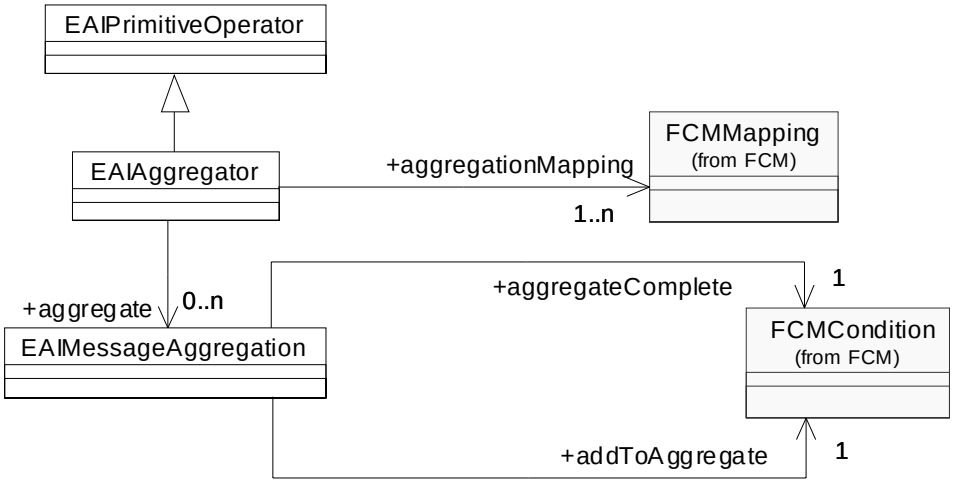


Figure 6-1 EAIAggregator

6.4.1.7 EAIRouterUpdate and EAIRouter

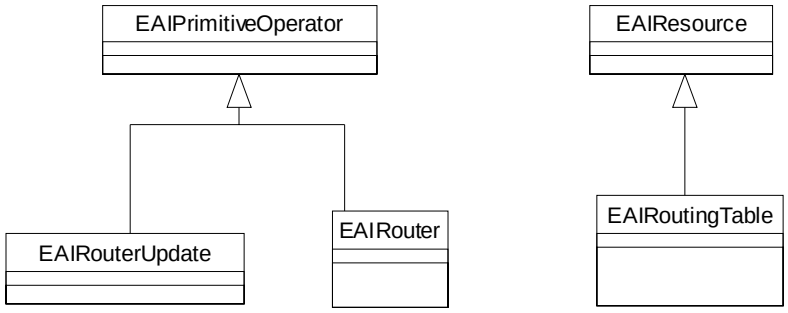


Figure 6-1 EAIRouter and EAIRouterUpdate

An EAIRouter routes a message to destinations listed in an EAIRoutingTable, which is maintained by EAIRouterUpdate. An EAIRoutingTable is a kind of EAIResource. An EAIRouter and an EAIRouterUpdate must each be associated with a single resource, which is an EAIRoutingTable.

An EAIRouter is a primitive operator with a single input terminal ("in") and a single output terminal ("out"). The target terminals of any EAILinks connected to the output terminal of an EAIRouter are added to the EAIRoutingTable for that EAIRouter as the initial set of routing targets. This set may be changed by the operation of an EAIRouterUpdate operator. When a message is received on the input terminal of an EAIRouter, dynamic EAILinks are established between the output terminal of the EAIRouter and each of the terminals in the current set of routing targets of the EAIRoutingTable of the EAIRouter. The input message is then copied to the output terminal and thus sent to each of the routing targets.

An EAIRouterUpdate is a primitive operator with a single input terminal ("control") and no output terminals. It expects to receive a message that conforms to the EAIRouterUpdateFormat content type. Such a message can specify either the addition (adds) or removal (removes) of a single terminal from the routing table that is associated with the operator as a resource.

6.4.1.8 EAISubscriptionOperator

An EAISubscriptionOperator is a subclass of EAIPrimitiveOperator with a single input terminal ("subscribe") and no output terminals. It expects an EAISubscriptionFormat as input. On receipt of an EAISubscriptionFormat, it adds information on the specified subscription to an EAISubscriptionTable.

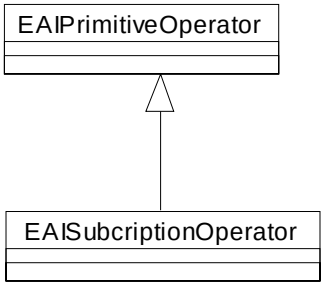


Figure 6-1 SubscriptionOperator

A message that conforms to the EAISubscriptionFormat specifies a target EAITerminal and a set of EAISubscriptionRules. In Figure 6-2, this is shown as a pair of derived associations. This indicates that the target and associated subscription rules can be computed from the message content.

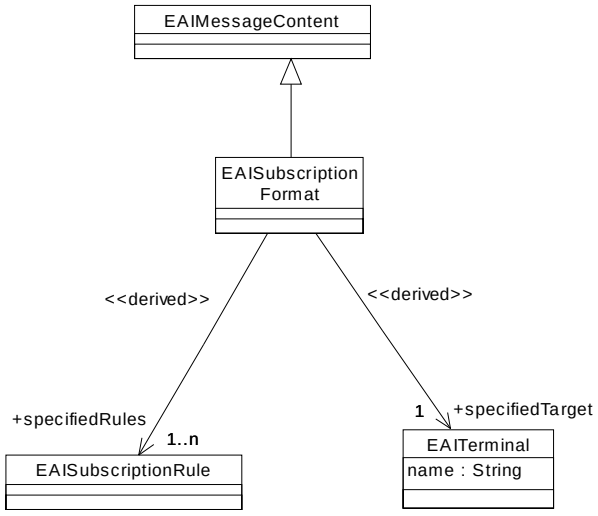


Figure 6-2 EAISubscriptionFormat

An EAISubscriptionTable is an EAIResource that is used to record the subscriptions received by an EAISubscriptionOperator. An EAISubscriptionOperator is an EAIPrimitiveOperator, which may have attached resources (see Figure 6-1). An EAISubscriptionOperator is specifically required to have exactly one resource, which must be an EAISubscriptionTable. An EAIPublicationOperator (see Section 6.4.1.9) referencing the same EAISubscriptionTable may then forward to subscribed target terminals messages that satisfy the subscription rules for those terminals.

An EAISubscriptionFilter is a subclass of EAIFilter. Its filterCondition is a set of EAISubscriptionRule

Constraints

An EAISubscriptionOperator has exactly one terminal, which is an input EAITerminal with the name “subscribe”.

The input terminal of an EAISubscriptionOperator is associated with an EAIParameter that has a message that is an EAISubscriptionFormat.

An EAISubscriptionOperator must have exactly one resource, which is an EAISubscriptionTable.

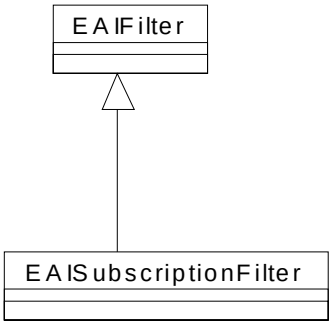


Figure 6-3 SubscriptionFilter

An EAISubscriptionRule has subclasses EAITopicRule and EAIContentRule. An EAITopicRule tests whether a message was published to one or more of an allowed set of topics, as recorded in the header for that message (see also Section 6.4.2.3, "Relationship between topic-based publishers and subscribers," on page 38). An EAIContentRule is a predicate that operates on the content of a message.

The filterCondition of an EASubscriptionFilter is an EAISubscriptionRule.

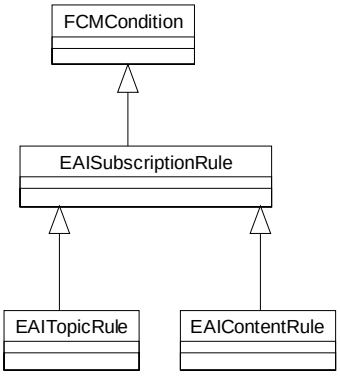


Figure 6-4 EAISubscriptionRule, EAITopicRule and EAIContentRule

6.4.1.9 EAIPublicationOperator

Description

The EAIPublicationOperator is used to model the publishing portion of the publish/subscribe more of information sharing. It forwards messages to target terminals recorded in the EAISubscriptionTable attached to it as a resource, if the messages meet the relevant subscription rules.

An EAIPublicationOperator is an EAIPrimitiveOperator with a single input terminal (“in”) and a single output terminal (“out”). When a message arrives at the input terminal, the EAISubscriptionRules for all subscriptions in the current state of the EAISubscriptionTable are evaluated on the message. For each subscription for which the rule is true, a dynamic, temporary EAILink is effectively established from the output terminal to the subscriber EAITerminal from the subscription. The input message is then copied to the output terminal and thus distributed to each subscriber.

If the target terminal of a dynamic EAILink is *not* an EAIQueuedInputTerminal, then the dynamic EAILink is considered to have *synchronization = unspecified*.The published message is simply placed on the identified target terminal. However, if the identified target terminal *is* an EAIQueuedInputTerminal (see Section 6.3.8), then the dynamic EAILink is considered to have *synchronization = asynchronous* and the published message is placed on the inputQueue of the target terminal.

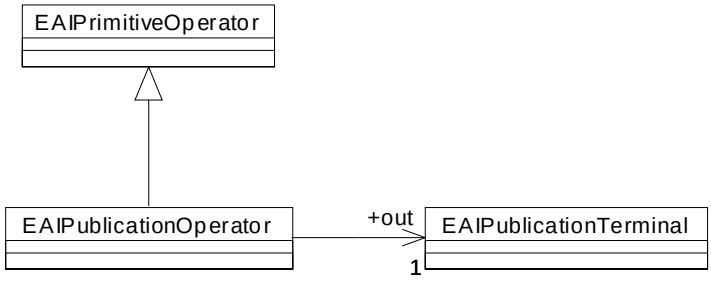


Figure 6-1 EAIPublicationOperator and EAISubscriptionOperator

Constraints

An EAIPublicationOperator has exactly one input terminal, which is an EAITerminal with the name “in”, and exactly one output terminal, which is an EAITerminal with the name “out”.

The messages of the EAIParameters associated with the two terminals of an EAIPublicationOperator must be the same.

An EAIPublicationOperator must have exactly one resource, which is an EAISubscriptionTable.

An EAIPublicationOperator has exactly one input terminal, which is an EAITerminal with the name “in”, and exactly one output terminal, which is an EAITerminal with the name “out”.

The messages of the EAIParameters associated with the two terminals of an EAIPublicationOperator must be the same.

6.4.1.10 EAITimeSetOperator

The TimeSetOperator is a subclass of EAIPrimitiveOperator, with a single input terminal ("set") and no output terminals. On receipt of a message, which must be specified by an EAITimerConditionFormat, it addes the timer and message applicability conditions given by the message to the list of conditions stored in the EAITimerConditionTable that is attached to it as a resource.

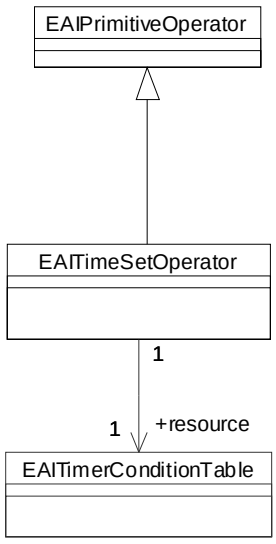


Figure 6-1 TimeSetOperator

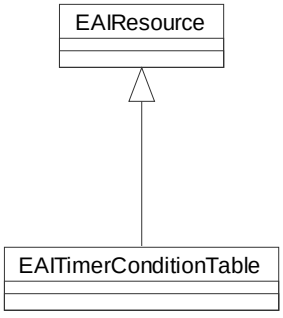


Figure 6-2 TimerConditionTable

A message in EAITimerConditionFormat is composed of two FCMConditions:

- timerCondition specifies a deadline (a time constraint). This may be relative or absolute.

- correlationCondition specifies the messages to which the timerCondition applies. This is often a condition on an element of a message header, such as the commonly used 'correlation identifier.'

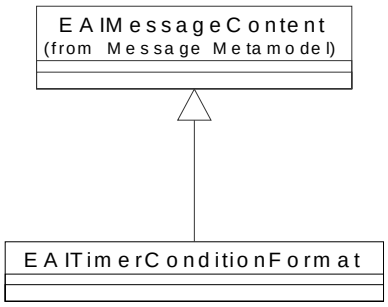


Figure 6-3 EAITimerConditionFormat

Constraints

Messages received on the “set” terminal must be in EAITimerConditionFormat.

6.4.1.11 EAITimeCheckOperator

EAITimeCheckOperator is a subclass of EAIPrimitiveOperator with a single input terminal ("check") and three output terminals ("ontime", "expiry" and "late"). On receipt of a message, it examines its set of conditions stored in the EAITimerConditionTable that is attached to it as a resource. If there is a timer condition that applies to the message, it checks that the condition is actually met. If so, the message is passed to the "ontime" terminal; if not, it is passed to the "late" terminal.

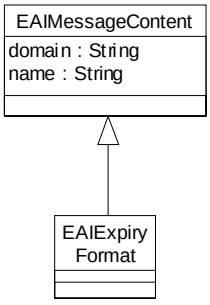


Figure 6-1 EAIExpiryFormat

At the time that a particular timer condition expires, a message of format EAIExpiryFormat is sent from the "expiry" terminal.

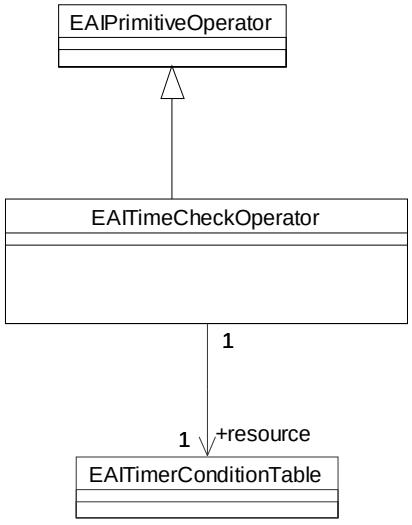


Figure 6-2 EAITimeCheckOperator

6.4.1.12 EAITimer

EAITimer is formed from a composition of EAITimeSetOperator and EAITimeCheckOperator.

It has two input terminals, "set" and "check," and two output terminals "out", "expiry" and "late", all of which map to terminals of the same name owned by the two primitive operators. Consequently, the "set" terminal causes the EAITimeSetOperator to be invoked, while messages sent to the "check" terminal cause the EAITimeCheckOperator to be invoked.

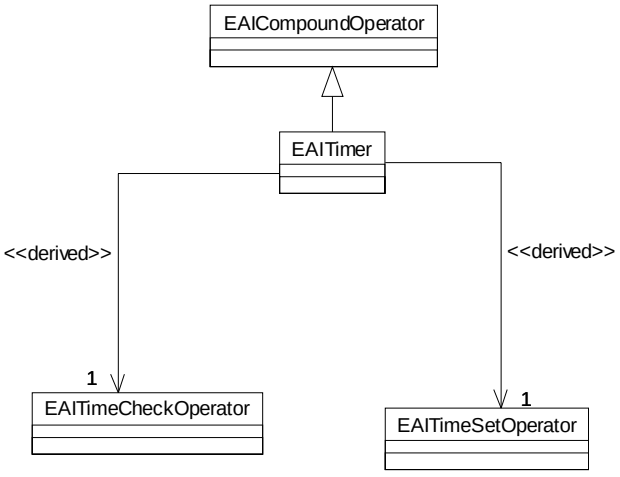


Figure 6-1

Constraints

The instance of EAITimeCheckOperator and EAITimeSetOperator from which an EAITimer is formed share the same EAITimerConditionTable.

6.4.2 Topic-based publish/subscribe

6.4.2.1 EAITopicPublisher

An EAITopicPublisher is a subclass of EAISource. It sends messages for publication to an EAIPublicationOperator. The set of topics that it publishes messages on is denoted by publishesOn. This is a derived association, since a topic publisher need not declare the set of topic it publishes on.

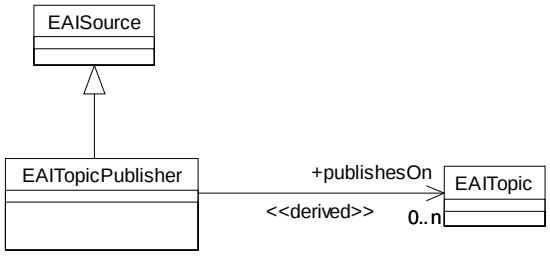


Figure 6-1

6.4.2.2 Topics 'allowed' by an EAITopicRule

An abstract representation of an EAITopicRule is the set of Topics that it allows.

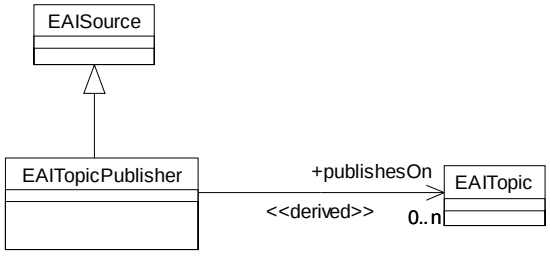


Figure 6-1 Topics allowed by an EAITopicRule

6.4.2.3 *Relationship between topic-based publishers and subscribers*

Topic-based publishers and subscribers are related to each other via the topics that they produce and consume.

For a input terminal representing a subscriber connected to a particular PublicationOperator, the set of topics it is interested in (subscribesTo) is determined by the topic which its filterCondition allows.

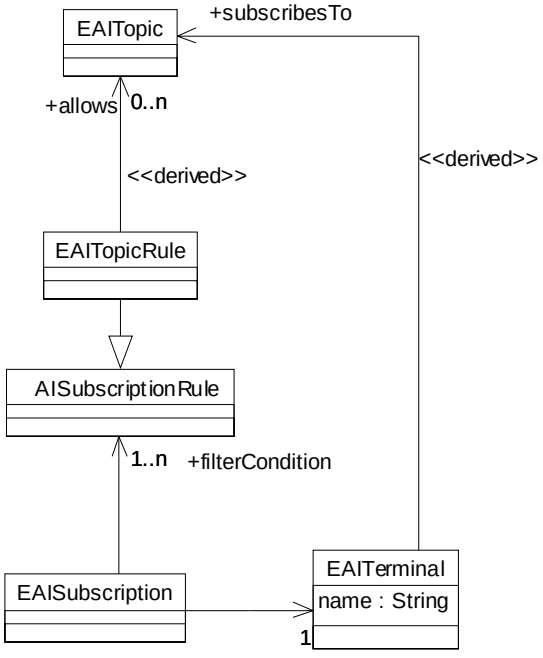


Figure 6-1 Relationship between a terminal and the topics for which it has a subscription

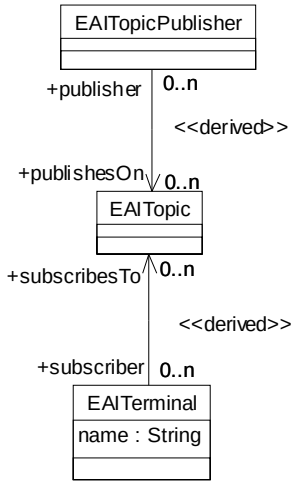


Figure 6-2 Relationship between publishers, subscribers and topics

6.5 CCA Component Library for EAI

This section specifies the CCA component library for EAI and mapping between EAI and CCA concepts. CCA provides for the modeling of collaboration similar to the EAI models in Chapters 8 and 9. The component library specifies the set of components required in CCA to represent the same concepts as the EAI meta model. By providing this component library and mapping between EAI and CCA users may transform models between EAI and CCA tools, integrating EAI systems with collaborations modeled with CCA. This information may be used by EAI or CCA tool vendors to automate such transformation and integration or may be used directly by users in a manual process.

For each of the listed EAI model elements a corresponding library component is defined. In each case the library component has the same name as the corresponding EAI model element.

6.5.1 Operators

6.5.1.1 EAIPrimitiveOperator

EAIPrimitiveOperator corresponds to an unconstrained CCA ProcessComponent.

The Terminal of the EAIPrimitiveOperator corresponds to Port of the CCA ProcessComponent.

Input Terminal corresponds to a CCA FlowPort with metaattribute direction = responds.

Output Terminal corresponds to a CCA FlowPort with metaattribute direction = initiates.

The handled ContentFormat of a Terminal in the EAIPrimitiveOperator corresponds to the type DataElement of the CCA FlowPort.

The Choreography of the CCA ProcessComponent corresponding to an EAIPrimiveOperator will have CCA PortActivity. This represents each CCA FlowPort corresponding to EAI input Terminal, followed by CCA Transition with target on CCA PortActivity that represents each CCA FlowPort corresponding to EAI output Terminal.

A CCA ProcessComponent, corresponding to an EAIPrimitiveOperator, can be utilized in a CCA Composition as a CCA ComponentUsage that uses the CCA ProcessComponent. For each CCA Port in the CCA ProcessComponent, there will be a CCA PortConnector corresponding to the CCA FlowPort of the used ProcessComponent.

In CCA, there is no fundamental distinction between primitive and non-primitive ProcessComponents. Rather, the "primitiveness" of a ProcessComponent is not externally observable. The CCA ProcessComponent may optionally have internal Composition detail, using other ProcessComponents.

In CCA, there is no fundamental distinction between primitive and non-primitive ProcessComponents. Rather, the "primitiveness" of a ProcessComponent is not externally observable. The CCA ProcessComponent may optionally have internal Composition detail, using other ProcessComponents.

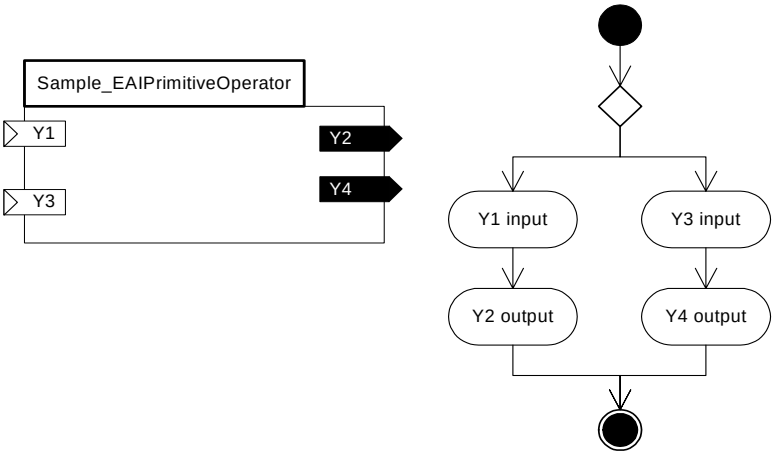


Figure 6-1 CCA notation for a sample generic EAIPrimitiveOperator

6.5.1.2 EAITransformer

EAITransformer is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with one CCA FlowPort with direction = responds and one CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAITransformer will show a CCA PortActivity on the FlowPort with direction = responds, followed by a CCA PortActivity on the FlowPort with direction = initiates.

The input and output CCA FlowPort will have different DataElement types. The ProcessComponent will transform from the input DataElement type to the output DataElement type.

The transformation to be performed on the DataElement contents can be specified in a Property of the CCA ProcessComponent as an expression, script or transformation specification in any of the transformation languages available. Alternatively, the transformation can be delegated into usages of other technology-specific transformation ProcessComponents in the internal Composition or into EAI transformer implementations.

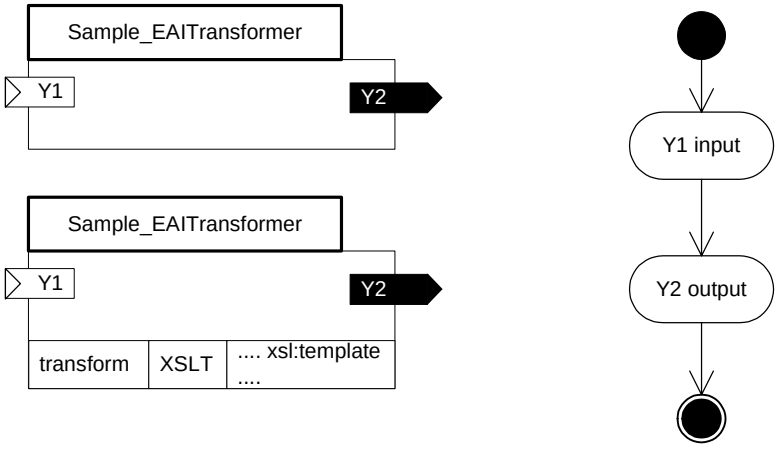


Figure 6-1 CCA notation for sample EAITransformer

6.5.1.3 EAIFilter

EAIFilter is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with one CCA FlowPort with direction = responds and two CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAIFilter will show a CCA PortActivity on the FlowPort with direction = responds, followed by a choice vertex, followed by a CCA PortActivity on each of the FlowPort with direction = initiates.

The input and each output CCA FlowPort will have the same DataElement type.

The criteria for the choice of true or false output terminal Port can be specified in a Property of the CCA ProcessComponent as an expression in any of the languages available. Criteria logic can also be delegated into usages of other ProcessComponents in the internal Composition.

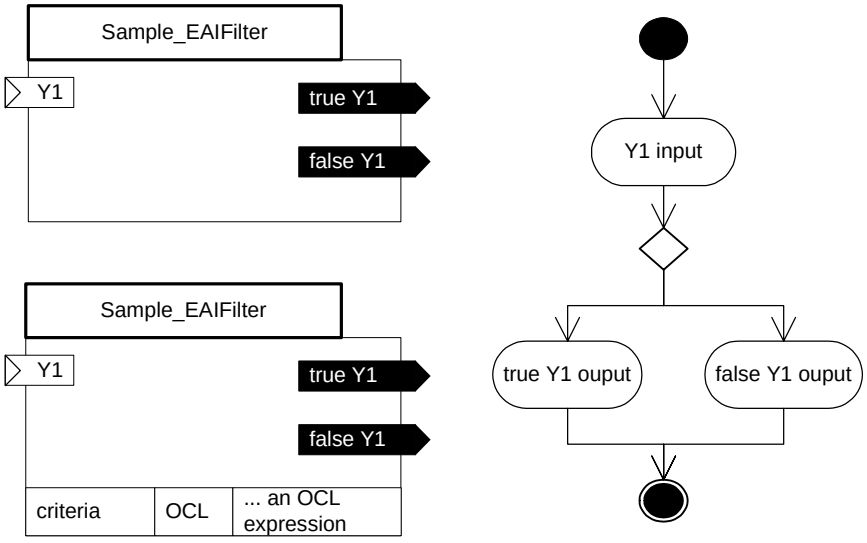


Figure 6-1 CCA notation for a sample EAIFilter

6.5.1.4 EAIStream

EAIStream is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds and a single CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAIStream will show a CCA PortActivity on the FlowPort with direction = responds, followed by a Fork, followed by CCA PortActivity on the FlowPort with direction = initiates, followed by a Join.

The input and output CCA FlowPort will have the same DataElement type. The ProcessComponent will store inputs to be sent later, possibly in a different order, through the output terminal FlowPort.

The algorithm used to determine when, and in which order, the incoming messages will be posted in the output terminal FlowPort can be specified as a Property of the EAIStream component, or it can be delegated into usages or other ProcessComponents in the internal Composition.

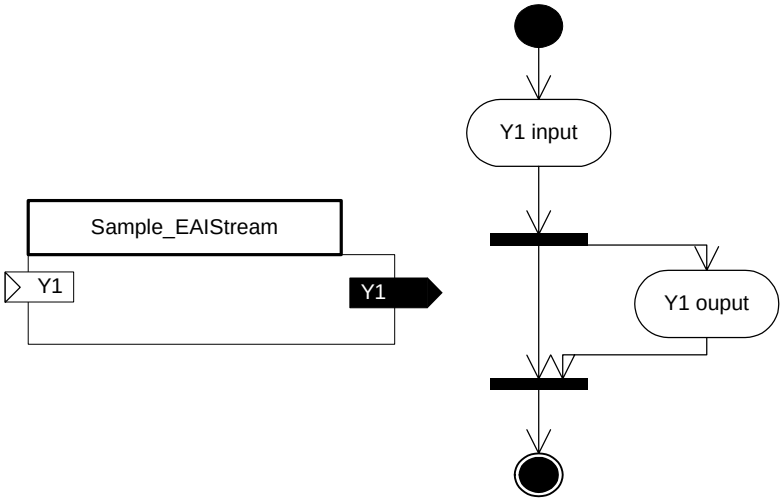


Figure 6-1 CCA notation for a sample EAIStream

6.5.1.5 EAICompoundOperator

EAICompoundOperator corresponds to an unconstrained CCA component. It will use other EAI Operator or Adapter ProcessComponents in the internal Composition.

The ProcessComponent for EAICompoundOperator will have externally connectable Ports that will be delegated into Ports of the internally used ProcessComponent.

Incoming messages on the external Port of the EAICompoundOperator ProcessComponent will be delivered to the internally connected Port of the ProcessComponent operators and adapters used.

Outgoing messages from the internally connected Port of the used ProcessComponent operators and adapters will be forwarded to the external outgoing Port of the EAICompoundOperator ProcessComponent.

This recursive composition capability of CCA corresponds to FCM and EAI recursive composition of nodes, operators and adapters.

For the user of an EAICompoundOperator ProcessComponent, there is no difference between using a Compound or a Primitive Operator. The internal composition of the Compound Operator remains encapsulated by the ProcessComponent. The user can only observe the external Port and Choreography of the ProcessComponent.

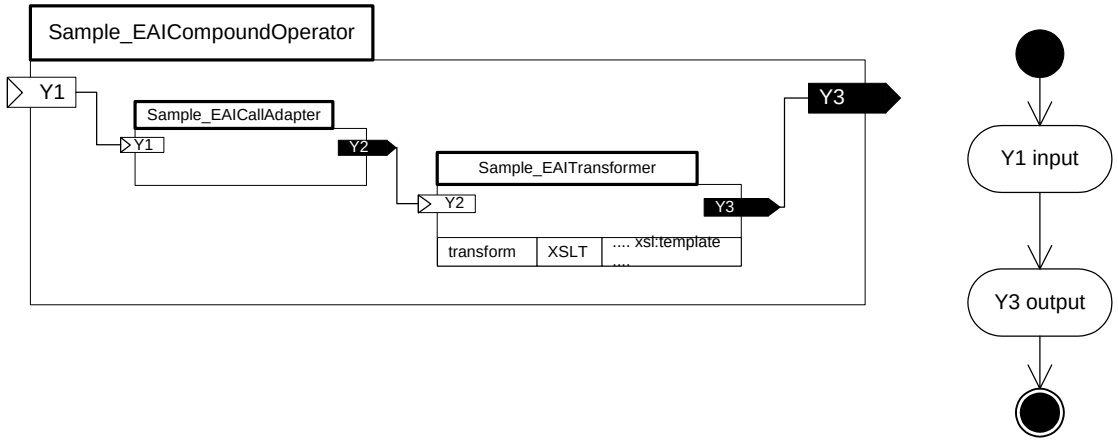


Figure 6-1 CCA notation for sample EAICompoundOperator

6.5.2 Adapters

6.5.2.1 EAISourceAdapter

EAISourceAdapter is a specialized ▵FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAISourceAdapter will show a CCA PortActivity on the FlowPort with direction = initiates.

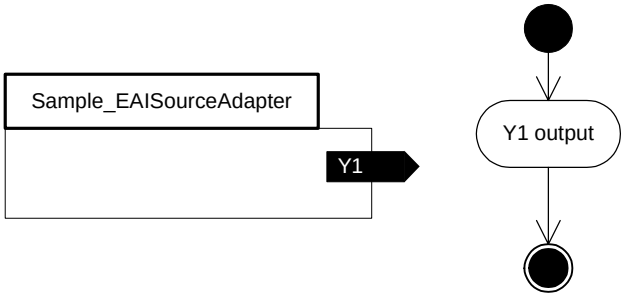


Figure 6-1 CCA notation for a sample EAISourceAdapter

When the EAISourceAdapter is to be utilized in Pull mode, an additional FlowPort will respond to a generic "Get" message that will trigger retrieval from the system and initiate the output.

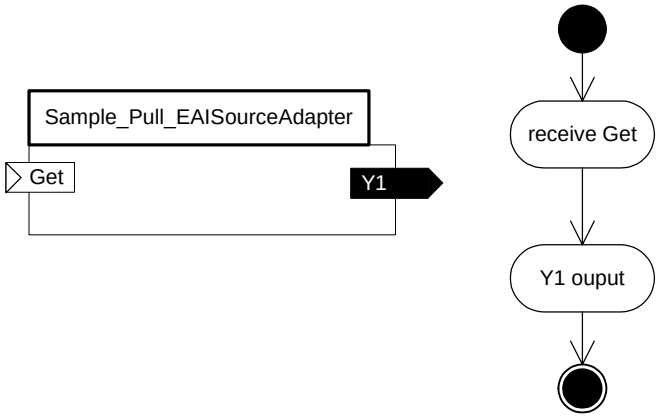


Figure 6-2 CCA notation for a sample Pull mode EAISourceAdapter

6.5.2.2 EAITargetAdapter

▵EAISourceAdapter is a specialized ▵FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds.

The Choreography of the CCA ProcessComponent corresponding to an EAITargetAdapter will show a CCA PortActivity on the FlowPort with direction = responds.

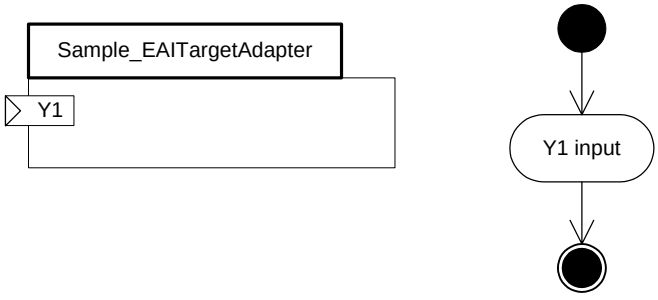


Figure 6-1 CCA notation for a sample EAITargetAdapter

6.5.2.3 EAIQueuedTargetAdapter

△EAISourceAdapter is a specialized △FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds.

An EAIQueuedTargetAdapter offers the same externally observable contract as the EAITargetAdapter but with different internal behavior, namely, queued delivery of messages to the system.

Queueing of messages can be directly implemented or delegated into usages of technology-specific message-queue ProcessComponents in the internal composition.

6.5.2.4 EAICallAdapter

△EAISourceAdapter is a specialized △FCMFunction. It corresponds to a CCA ProcessComponent with a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates.

Alternatively, an EAICallAdapter may correspond to a CCA ProcessComponent with a ProtocolPort, with subPorts obeying a Protocol having a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates. This aggregation in a single ProtocolPort of the FlowPorts for the call and response messages provides as single connection point for the full call-response, which is similar to the conventional functional invocation in programming languages.

The Choreography of the CCA ProcessComponent corresponding to an EAICallAdapter will show a CCA PortActivity on the FlowPort with direction = responds, followed by a CCA PortActivity on the FlowPort with direction = initiates.

An EAICallAdapter accepts synchronous calls that are not externally observable. It converts these to asynchronous messages that are sent on the output terminal initiating FlowPort. It receives a response on the input terminal responding FlowPort and passes an equivalent response to the caller. The EAICallAdapter must implement the logic and mechanisms to wait for the asynchronous response and rebind to the thread of the calling process.

The input and output CCA FlowPort may have the same or different DataElement type. The ProcessComponent will convert the input to the type required by the system. The system will respond with information of a certain type that the ProcessComponent must convert into the output DataElement type.

The transformation to be performed on the DataElement contents can be specified in Properties of the CCA ProcessComponent as an expression, script or transformation specification in any of transformation languages available. Alternatively, the transformation can be delegated into usages of other technology-specific transformation ProcessComponents in the internal Composition.

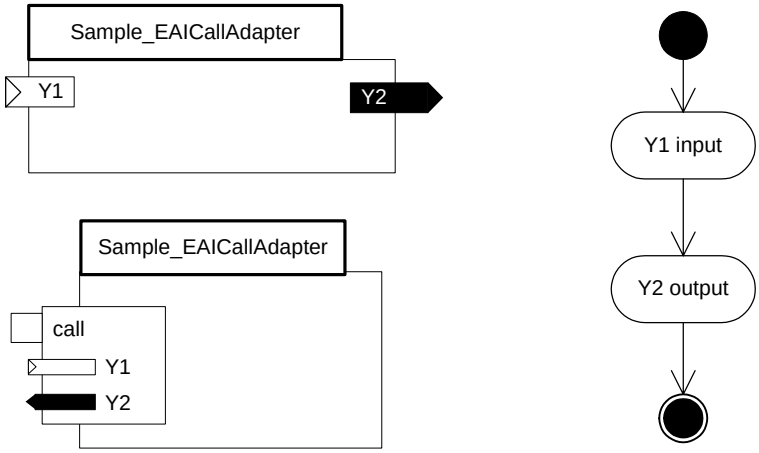


Figure 6-1 CCA notation for sample EAICallAdapter

6.5.2.5 *EAIRequestReplyAdapter*

▵EAISourceAdapter is a specialized ▵FCMFunction. It corresponds to a CCA ProcessComponent with a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates.

Externally, an EAIRequestReplyAdapter exposes similar contract and behaves like the EAICallAdapter.

The EAIRequestReplyAdapter accepts asynchronous messages. It invokes a system synchronously and returns the response as a message that other applications can process asynchronously. The RequestReplyAdapter presents an asynchronous interface on a synchronous invocation.

6.5.3 *CCA and EAI Metamodel Mapping Tables*

The following table shows the mapping between EAI and CCA model elements. In many cases the EAI library component is also part of the mapping.

Table 6-1 Model elements mapping table

EAI metamodel element	**CCA metamodel element**	**Library Component (Component Used)**
EAIFlow	ProcessComponent	
EAIRouterComposition	ProcessComponent	
EAIPrimitiveOperator	ComponentUsage	EAIPrimitiveOperator
EAICompoundOperator	ComponentUsage	EAICompoundOperator
EAITargetAdapter	ComponentUsage	EAITargetAdapter

Table 6-1 Model elements mapping table

EAI metamodel element	CCA metamodel element	Library Component (Component Used)
EAISourceAdapter	ComponentUsage	EAISourceAdapter
EAICallAdapter	ComponentUsage	EAICallAdapter
EAIRequestReplyAdapter	ComponentUsage	EAIRequestReplyAdapter
EAIFilter	ComponentUsage	EAIFilter
EAIStream	ComponentUsage	EAIStream
EAIPostDater	ComponentUsage	EAIPostDater
EAITransformer	ComponentUsage	EAITransformer
EAIDBTransformer	ComponentUsage	EAIDBTransformer
EAIAggregator	ComponentUsage	EAIAggregator
EAIRouter	ComponentUsage	EAIRouter
EAIRouter	ComponentUsage	EAIRouter
EAIRouterUpdate	ComponentUsage	EAIRouterUpdate
EAISubscriptionOperator	ComponentUsage	EAISubscriptionOperator
EAISubscriptionFilter	ComponentUsage	EAISubscriptionFilter
EAIPublicationOperator	ComponentUsage	EAIPublicationOperator
EAITimeSetOperator	ComponentUsage	EAITimeSetOperator
EAITimeCheckOperator	ComponentUsage	EAITimeCheckOperator
EAITimer	ComponentUsage	EAITimer
EAISource	Port with direction = responds	
EAIQueuedSource	Port with direction = responds	
EAITopicPublisher		
EAISink	Port with direction = initiates	
EAIQueuedSink	Port with direction = initiates	
EAILink	Connection	
EAIMessageOperation	FlowPort or OperationPort	
EAITerminal	PortConnector	
EAIQueuedInputTerminal	PortConnector	
EAIQueuedOutputTerminal	PortConnector	
EAIPublicationTerminal		
EAISubscriptionRule		
EAITopicRule		
EAIContentRule		
EAIMessageTimerCondition		

Table 6-1 Model elements mapping table

EAI metamodel element	CCA metamodel element	Library Component (Component Used)
EAIMessageContent	CompositeData	
EAIExceptionNotice	CompositeData	
EAIRequestFormat		
EAIQueue		
EAIContent		
EAIRouterUpdateFormat		
EAIAddTargetFormat		
EAISubscriptionFormat		
EAIResource		
EAIMessageAggregation		
EAISubscription		
EAITopic		

Examples of the CCA modeling elements are presented in Chapter 11.

EAI Common Application Metamodel 7

7.1 Business Requirements and Value

The current trend for new applications is to embrace open Web standards that simplify construction and scalability. As new applications are built, it is crucial to integrate seamlessly with existing systems while introducing new business models and new business processes.

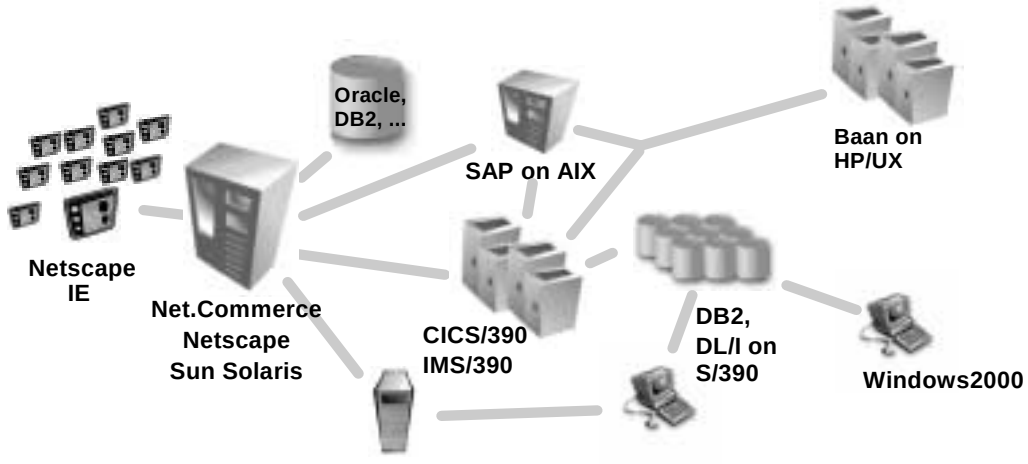


Figure 7-1 Multiple Application and Development Environments

Analysts from the Meta Group estimate that more than 70 % of corporate data lives on the mainframe, much of that on the S/390. Many transactions may be initiated by a Windows/NT or Unix server, but they will be completed on the mainframe under applications, such as CICS, or IMS applications. It is important to leverage and reuse these existing assets, including stored procedures, to provide interoperability with existing applications.

The above figure depicts multiple application components with multiple development teams and environments. Where is the application in this picture? Everywhere! How is the application assembled? With connectors!

Connectors are a central part of the application framework for e-business. The demand is to connect to anything interesting as quickly, and as easily, as possible.

A connector is required to match the interface requirements of the adapter and the legacy application. It is also required to map between the two interfaces. Standardized metamodels for application interfaces allow reuse of information in multiple connector tools. It will not only reduce work to create a connector, but also reduce work needed to develop connector builder tools, thus an incentive to connector suppliers.

7.2 Common Application Metamodel for Applications Interfaces

Business integration technology requires connectors to provide interoperability with existing applications. Connectors support leveraging and reuse of data and business logic held within existing application systems. The job of a connector is to connect from one application system server "interface" to another; it is not meant for an individual application program. Therefore, an application-domain interface metamodel describes signatures for input and output parameters and return types for a given application system domain (e.g. IMS, MQSeries); it is not for a particular IMS or MQSeries application program. The metamodel contains both syntactic and semantic interface metadata.

The following figure showing the EAI metamodel for application interfaces enables integration of application components into event-based messaging model including Flow models.

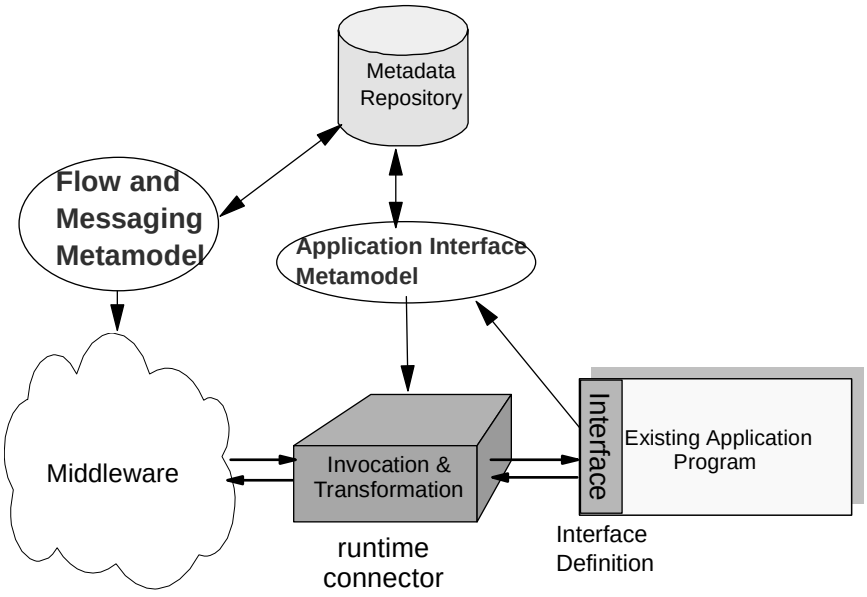


Figure 7-2 Application Interface Metamodel

The flow and messaging middleware invokes applications through the application interfaces. These interfaces are the access points to the applications through which all input and output is connected to the middleware. The interfaces are described in terms of the Application Interface Metamodels. Transformation processing according to the metamodel could take place in source/client applications, target applications, or a gateway.

7.2.1 End-to-End Connector Usage Using EAI Common Application Metamodel

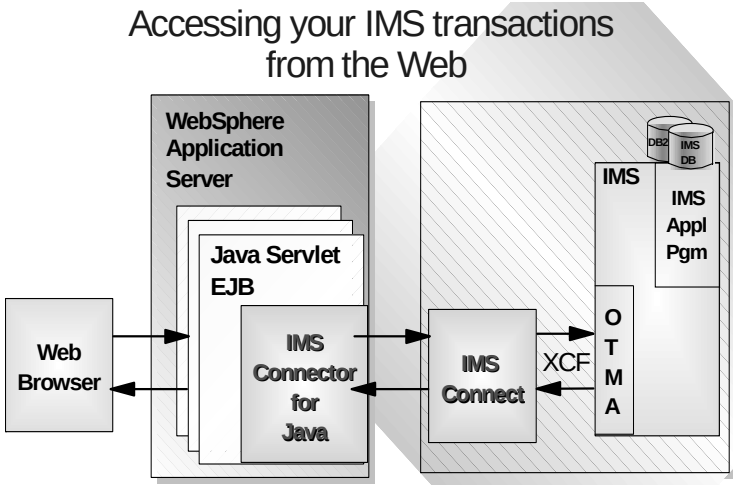
The EAI Common Application Metamodel (CAM) consists of meta-definitions of message signatures, independent of any particular tool or middleware. Different connector builder tools can use this information to ensure the "handshaking" between these application programs, across different tools, languages, and middleware. For example, if you have to invoke an MQSeries application, you would need to build a MQ message using data from a GUI tool and deliver it using the MQ API. Similarly, when you receive a message from the MQSeries application, you would need to get the buffer from MQSeries, parse it and then put it into a GUI tool data structure. These functions can be designed and implemented efficiently by a connector builder tool using EAI CAM as standardized metamodels for application interfaces.

EAI CAM can be populated from many sources, including copy books, to generate HTML forms and JavaServer Page (JSP) for gathering inputs and returning outputs. An example of a connector as depicted in the previous figure is that the flow and message middleware makes a function call to an enterprise application by calling the connector

that then calls the enterprise application API. The connector does language and data type mappings, for example, to translate between XML documents and COBOL input and output data structures based on EAI CAM. Connectors and EAI CAM provide the end-to-end integration between the middleware and the enterprise applications.

Using IMS as an example: Let's say that you must pass an account number to an IMS transaction application program from your desktop to withdraw $50.00. With EAI CAM and a connector builder tool, you will first generate an input HTML form and an output JSP; and develop a middleware code necessary to support the request. The desktop application fills the request data structure (i.e. an input HTML form) with values and calls the middleware. The middleware service code will take the data from the GUI tool, build an IMS Connect XML-formatted message, and deliver the message to the IMS gateway (i.e. IMS Connect) via TCP/IP. IMS Connect translates between the XML documents and the IMS message data structures in COBOL using the metadata definitions captured in EAI CAM. It then, in turn, sends the IMS message data structures to IMS via Open Transaction Manager Access (OTMA). The IMS COBOL application program runs, and returns the output message back to the middleware service code via IMS Connect. The middleware service code gets the message and populates the output JSP page (i.e. previously generated GUI tool reply data structures) with the reply data. The transaction output data will then be presented to the user.

IMS Connect and IMS OTMA are connector products that enable applications to interact with systems outside the host machine. For example, IBM Connect allows IMS to exchangedata with sources outside z/Series using TCP/IP. IBM's WebSphere Studio is an example of a 'connector builder tool.' Once the connector builder tool has generated a servlet and/or transformer code the for the applicaiton, the code can be deployed on a web server such as IBM WebSphere Application Server to communicate with the backend application via connectors such as IBM Connect and OMS OTMA. Below is a picture to help explain.



7.3 Common Application Metamodel

CAM is a group of interface metamodels that consist of enterprise application interface metamodels, language metamodels and physical representation metamodels. These include C, C++, Java, COBOL, PL/I, Type Descriptor, TDLang, IMS transaction messages, IMS MFS, and CICS BMS, etc. Note that the Java metamodel is defined in the OMG EDOC (Enterprise Distributed Object Computing) submission.

CAM is highly reusable and independent of any particular tool or middleware. CAM is an incentive to connector suppliers. It reduces work to create and develop connector and/or connector-builder tools. With CAM, tools can now easily access enterprise applications, e.g. IMS and CICS applications; and tools can also access any CAM enabled applications. CAM is used to describe information needed to easily integrate applications developed in common programming models with other systems. CAM can be used for both synchronous and asynchronous invocations.

Because CAM also provides physical representation of data types and storage mapping to support data transformation in an enterprise application integration environment, it enables Web services for enterprise applications.

In a nutshell, CAM is needed for

- connector and/or connector-builder tools (Development time)
- data transformation in an enterprise application integration environment (Execution time)
- data type mapping between mixed languages
- data translations from one language and platform domain into another
- data driven impact analysis for application productivity and quality assurance
- viewing of programming language data declarations by developers

CAM uses MOF and UML class modeling mechanisms. Every CAM class in an instance of a MOF class at the M2 level.

7.3.1 Enterprise Application Interface Metamodels

The Enterprise Application Interface metamodel describes signatures for input and output parameters and return types for application system domains.

The Enterprise Application Interface Metamodels listed as follows are non-normative and can be found in Appendix A.

- IMS Transaction Message
- IMS MFS
- IMS CICS BMS

7.3.2 Language Metamodels

The language metamodel, e.g. COBOL metamodel, is used by enterprise application programs to define data structures (semantics) that represent connector interfaces. An association between language metamodels (semantics) and the physical layout metamodel (syntactic) is necessary in order for the marshaller to correctly format the byte string. This association between language metamodels and Type Descriptor metamodel is further detailed in Section 7.3.9, “Physical Representation Model: TDLang Interaction Diagram,” on page 7-23. It is important to connector developers that connector tools show the source language, the target language, and the mapping between the two languages. The CAM language metamodel also includes the declaration text in the model that is not editable (i.e., read-only model). Because the connector/adapter developer would probably prefer to see the entire COBOL data declaration, including comments and any other documentation that would help him/her understand the business role played by each field in the declaration.

The language metamodel is also to support data driven impact analysis for application productivity and quality assurance. (But, it is not the intention of the CAM to support reproduction of copybooks.)

The language metamodels describing application interface data are listed as follows:

- C
- C++
- COBOL
- PL/I
- Java (Java metamodel is in the OMG EDOC final submission document.)

These language metamodels are found in Chapter 13.

7.3.3 Physical Representation Model: TDLang Metamodel

The TDLang metamodel serves as base classes to CAM language metamodels by providing a layer of abstraction between the Type Descriptor metamodel and any CAM language metamodel, including higher level languages. All TDLang classes are abstract and common to all the CAM language metamodels. All associations between TDLang classes are marked as "volatile," "transient," or "derived" to reflect that the association is derived from the language metamodel. The TDLang model does not provide any function on its own, but it is the type target for the association from the Type Descriptor metamodel to the language metamodels.

With the TDLang base classes, the Type Descriptor metamodel can be used as a recipe for runtime data transformation (or marshaling) with the language-specific metamodel for overall data structures and field names, without duplicating the aggregation (parent-child) associations present in the language model.

The TDLang model eliminates the need to have unique associations from each language model to the Type Descriptor model (e.g., cobolToTD and cToTD). All language models can access InstanceTDBase by calling the instanceTDBase association through the parent TDLangElement class.

The following figure illustrates the TDLang Metamodel. TDLang connects language models to the Type Descriptor Model. The TDLang metamodel acts as a generic placeholder for a variety of language models to inherit from. Following the diagram is a brief explanation of what each class represents.

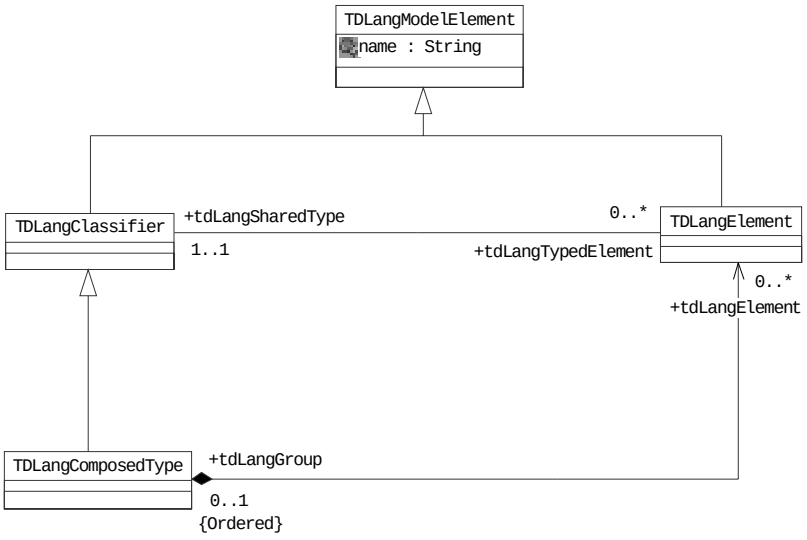


Figure 7-6 TDLang Metamodel

7.3.4 TDLang Metamodel Descriptions

7.3.4.1 TDLangClassifier

TDLangClassifier is the parent class of all CAM language Classifier classes and TDLangComposedType. TDLangClassifier represents all data types of a CAM language metamodel. Since TDLangClassifier is abstract, it is implemented by language specific classifier classes. Sample subclasses of TDLangClassifier include String, integer, character, float, and addressable pointers for each language model. Subclasses of TDLangClassifier provide the type information declared by a TDLangElement.

7.3.4.2 tdLangTypedElement : TDLangElement

Used by the classifier associated to an element within a ComposedType to navigate back to the parent ComposedType.

7.3.4.3 TDLangComposedType

TDLangComposedType represents the type of data with subcomponents. TDLangComposedType is the parent class of all CAM language ComposedTypes. Since TDLangComposedType is abstract, it is implemented by language specific composed classes. Sample subclasses of TDLangComposedType are COBOL 01-level data declarations with nested elements, C structs and unions, and PL/I structures, unions, or elementary variables and arrays.

tdLangElement : TDLangElement

Used by TDLangComposedType to get a list a TDLangElements contained within the composed type.

7.3.4.4 TDLangElement

TDLangElement is the most basic, fundamental core class of the TDLang Metamodel. TDLangElement is the parent class of all CAM language element classes. TDLangElement represents typed unit elements declared in a copybook or source code, that is typed data elements without a subcomponent. Since TDLangElement is abstract, it is implemented by language specific element classes. Sample subclasses of TDLangElement are COBOLElement, CTypedElement, and PLIElement.

tdLangGroup: TDLangComposedType

Used by TDLangElement to determine the TDLangComposedType it belongs to.

tdLangSharedType : TDLangClassifier

Used by TDLangElement to determine the type associated to the element.

7.3.4.5 TDLangModelElement

TDLangModelElement is the parent class of all TDLang classes. Each instance of TDLangModelElement represents either a declared element or a classifier type. TDLangModelElements that represent a declared element can refer to another instance of TDLangModelElement that represents a classifier type. Since elements and user-defined types may have associated names, TDLangModelElement has a name attribute that can be separately instatiated by TDLangElement and TDLangClassifier.

7.3.5 Physical Representation Model: Type Descriptor Metamodel

Type Descriptor metamodel presents a language and platform independent way of describing implementation types, including arrays and structured types. This information is needed for marshaling and for connectors that have to transform data from one language and platform domain into another. Inspections of the type model for different languages can determine the conformance possibilities for the language types. For example, a `long` type in Java is often identical to a binary type (`computational-5`) in COBOL, and if so, the types may be inter-converted without side effect. On the other

hand, an alphanumeric type in COBOL is fixed in size and if mapped to a Java type, loses this property. When converted back from Java to COBOL, the COBOL truncation rules may not apply, resulting in computation anomalies. In addition, tools that mix languages in a server environment (e.g., Java and COBOL in CICS and IMS) should find it useful as a way to determine how faithfully one language can represent the types of another. Therefore, an instance of the Type Descriptor metamodel describes the physical representation of a specific data type for a particular platform and compiler. The following figures illustrate the classes that constitute the Type Descriptor metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

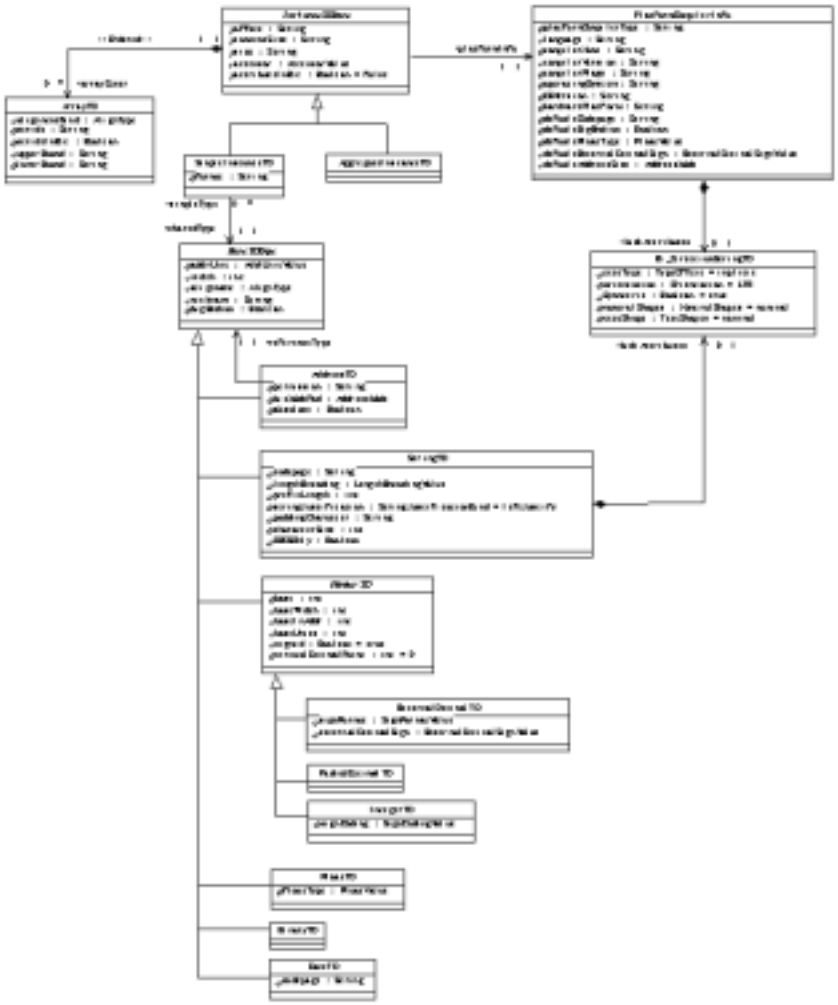


Figure 7-1 Type Descriptor metamodel

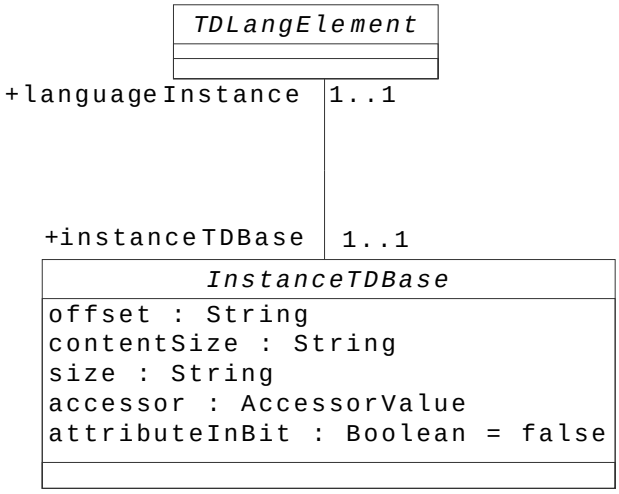


Figure 7-2 TDLang to Type Descriptor

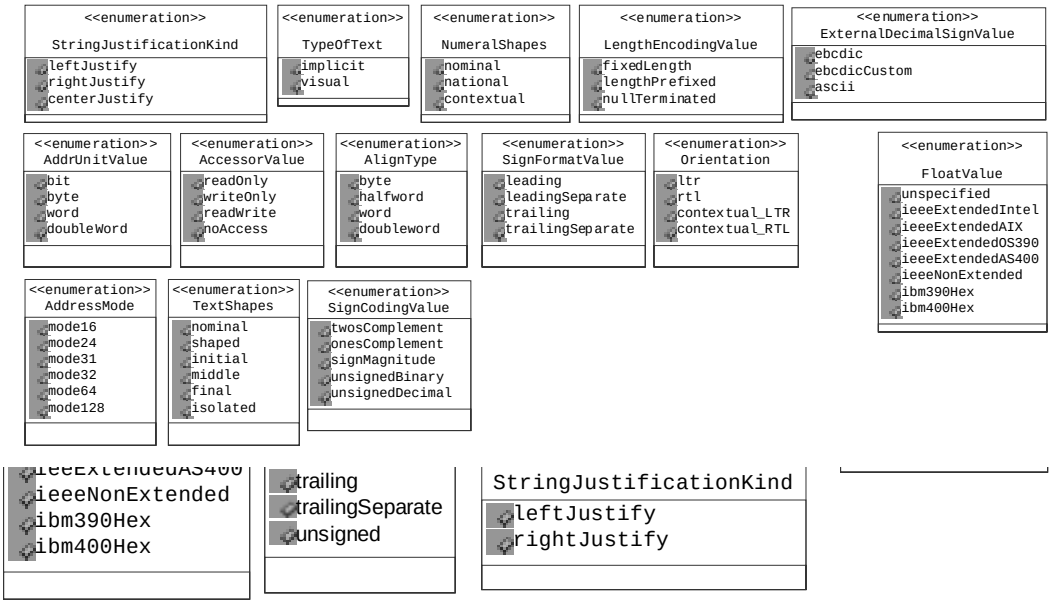


Figure 7-3 Type Descriptor Enumerations

7.3.6 Type Descriptor Metamodel Descriptions

7.3.6.1 AddressTD

AddressTD represent pointers/addresses. Addresses should be considered to be different from NumberTD class because some languages on certain machines (e.g., IBM 400) represent addresses with additional information, such as permission type (which is not represented in NumberTD class)

7.3.6.2 ArrayTD

ArrayTD holds information for array types. Data element instances may be defined as repeating groups or arrays. This is modeled as a one-to-many association between InstanceTDBase and the ArrayTD model type. One instance of ArrayTD is created for each dimension, subscript, or independent index of the data element. Each instance holds information about the bounds and accessing computations. The association order between ArrayTD and InstanceTDBase is the same as the order for the corresponding association in the language model, and reflects the syntactic ordering of the indices as defined by the programming language.

7.3.6.3 BaseTDType

BaseTDType is the abstract parent class of all types in the TD Metamodel. BaseTDType holds implementation information common to all data types of the same runtime environment, as specified by PlatformCompilerInfo.

7.3.6.4 Bi-DirectionStringTD

Bi-DirectionStringTD is a set of optional attributes contained by PlatformCompilerInfo and StringTD. Bi-DirectionStringTD represents strings with extended properties and formats such as numeral shapes and right-to-left reading direction. When Bi-DirectionStringTD is contained by PlatformCompilerInfo, the bi-directional attributes will apply to the entire application program, whereas when Bi-DirectionString is contained by StringTD, the bi-directional attributes will only apply the specified string element.

7.3.6.5 BinaryTD

BinaryTD represents a string of binary bits whose format is not to be modified.

7.3.6.6 DateTD

DateTD represents date types with its associated format (e.g., mm/dd/yyyy, dd/mm/yyyy)

7.3.6.7 ExternalDecimalTD

ExternalDecimalTD represents numbers expressed in external decimal format.

7.3.6.8 FloatTD

FloatTD represents floating point numbers declared by a language element.

7.3.6.9 InstanceTDBase

lnstanceTDBase is the most basic, fundamental core class of the Type Descriptor Metamodel. Every TD Metamodel instance contains at least one instance of InstanceTDBase. For each instance of a CAM language Element class there is a corresponding instance of InstanceTDBase. InstanceTDBase contains attributes that describe the physical layout of each declared variable and structure element in a program. It is an abstract class realized by either SimpleInstanceTD or AggregateInstanceTD. To find the parent of any instance (if it has one) navigate the association back to the CAM Language Element class (via a language-independent element class, e.g., TDLangElement), follow the association to the language-specific Composed class, then follow the association back to the parent InstanceTDBase.

7.3.6.10 IntegetTD

IntegerTD represents numbers expressed in binary format.

7.3.6.11 Number TD

NumberTD represents all integer and packed decimals. NumberTD is the parent class of ExternalDecimalTD, PackedDecimal, and IntegerTD.

7.3.6.12 PackedDecimalTD

PackedDecimalTD represents numbers expressed in packed decimal format.

7.3.6.13 PlatformCompilerInfo

PlatformCompilerInfo captures the static compiler and program runtime environment. Since this static information is shared by all instances of InstanceTDBase, this class only needs to be instantiated once.

7.3.6.14 SimpleInstanceTD and AggregateInstanceTD

Both SimpleInstanceTD and AggregateInstanceTD are subclasses of InstanceTDBase. InstanceTDBase has two concrete subtypes: SimpleInstanceTD and AggregateInstanceTD. SimpleInstanceTD models data elements without subcomponents, while AggregateInstanceTD models data elements with subcomponents. To find the subcomponents of an AggregateInstanceTD, one must

navigate back to the corresponding data element declaration in the CAM language model. There, the association between an aggregate type and its subcomponents may be navigated, leading to a set of subcomponent data elements, each of which has one or more corresponding instances in the Type Descriptor model.

7.3.6.15 StringTD

StringTD represents standard left-to-right format character strings. StringTD also supports single characters elements.

7.3.6.16 Type Descriptor Enumerations

- AccessorValue enumerates permission rights for each TDLangElement.
- AddressMode enumerates addressable units per hardware platform.
- AddrUnitValue enumerates the unit associated with the value of address attributes in Type Descriptor Metamodel.
- AlignType enumerates alignment delimiters for each data type specified by BaseTDType.
- ExternalDecimalSignValue enumerates various encoding methods to represent number characters.
- FloatValue enumerates floating types supported by Type Descriptor Metamodel.
- LengthEncodingValue enumerates string length encoding values supported by Type Descriptor Metamodel.
- NumeralShapes enumerates how Arabic numeric glyphic characters will be displayed on screen.
- Orientation enumerates how text should be presented from layout in memory.
- SignCodingValue enumerates numeric sign encoding values supported by Type Descriptor Metamodel.
- SignFormatValue enumerates the position of the positive and negative sign in a numeric item.
- StringJustificationKind enumerates string justification layout values supported by Type Descriptor Metamodel.
- TextShapes enumerates the shape of Arabic characters in relation to its position to neighboring characters.
- TypeOfText enumerates the method text should be read from memory. Text in memory can either be interpreted as it is logically implied by context or as it should be displayed visually on screen.

7.3.7 Type Descriptor Formulas

In the following discussion, “field” refers to a component of a language data structure described by the Type Descriptor metamodel, while “attribute” denotes part of the model, and has a value representing a “property” of the field. Thus the value of a field means a run-time value in a particular instance of a language data structure, whereas the value of an attribute is part of the description of a field in a language data structure, applies to all instances of the data structure, and is determined when the data structure is modeled.

For most attributes in an instance of the Type Descriptor metamodel, the value of the attribute is known when the instance is built, because the properties of the fields being described, such as size and offset within the data structure, are invariant. But if a field in a data structure is defined using the COBOL `OCCURS DEPENDING ON` construct or the PL/I `Refer` construct, then some properties of the field (and properties of other fields that depend on that field’s value) cannot be determined when the model instance is built.

Properties that can be defined using these language constructs are string lengths and array bounds. A property that could indirectly depend on these language constructs is the offset of a field within a structure, if the field follows a variable-size field.

In order to handle these language constructs, properties of a field that could depend on these constructs (and thus the values of the corresponding attributes), are defined with strings that specify a formula that can be evaluated when the model is used.

However, if a property of a field is known when the model instance is built, then the attribute formula simply specifies an integer value. For example, if a string has length 17, then the formula for its length is “17.”

The formulas mentioned above are limited to the following:

- Unsigned integers
- The following arithmetic integer functions

```
neg(x)   := -x          // prefix negate
add(x,y) := x+y         // infix add
sub(x,y) := x-y         // infix subtract
mpy(x,y) := x*y         // infix multiply
div(x,y) := x/y         // infix divide
max(x,y) := max(x,y)
min(x,y) := min(x,y)
mod(x,y) := x mod y
```

The mod function is defined as mod(x,y) = r where r is the smallest non-negative integer such that x-r is evenly divisible by y. So mod(7,4) is 3, but mod(-7,4) is 1. If y is a power of 2, then mod(x,y) is equal to the bitwise-and of x and y-1.

- The val function

  The val function returns the value of a field described by the model. The val function takes one or more arguments, and the first argument refers to the level-1 data structure containing the field, and must be either:
  - the name of a level-1 data structure in the language model
  - the integer 1, indicating the level-1 parent of the variable-size field. In this case, the variable-size field and the field that specifies its size are in the same data structure, and so have a common level-1 parent.

Here level-1 data structures refer to the top level declaration of a composed type element and a level-1 parent refers to 01 element that contains the field in question. An example of a level-1 data structure in COBOL would be an 01 element that contains other elements. The subsequent arguments are integers that specify the ordinal number within its substructure of the (sub)field that should be dereferenced.

By default, COBOL data fields within a structure are not aligned on type-specific boundaries in storage. For example, the “natural” alignment for a four-byte integer is a full-word storage boundary. Such alignment can be specified by using the `SYNCHRONIZED` clause on the declaration. Otherwise, data fields start immediately after the end of the preceding field in the structure. Since COBOL does not have bit data, fields always start on a whole byte boundary.

For PL/I, the situation is more complicated. Alignment is controlled by the Aligned and Unaligned declaration attributes. By contrast with COBOL, most types of data, notably binary or floating-point numbers, are aligned on their natural boundaries by default.

7.3.8 Type Descriptor Formula Examples

7.3.8.1 COBOL

The examples use the proposed inline comment indicator “*>” from the draft standard. It is not yet legal COBOL usage.

1. Consider the following data description:

```
*> Field                                Offset

   01 Used-Car.                        *> "0"
    02 Summary.                      *>    "0"
       03 Make pic x(36).            *>    "0"
       03 Model pic x(44).           *>    "36"
       03 VIN pic x(13).             *>    "80"
       03 Color pic x(10).           *>    "93"
         88 Red value 'Red'.
         88 White value 'White'.
         88 Blue value 'Blue'.
     02 History.                     *>    "103"
```

```
       03 Mileage pic 9(6).            *>     "103"
       03 NumClaims binary pic 9.      *>     "109"
       03 InsCode pic x.               *>     "111"
       03 Claims.                      *>     "112"
         04 Claim occurs 1 to 9 times
             depending on NumClaims.    *> stride(1) = "157"
           05 ClaimNo pic x(14).        *>     "112"
           05 ClaimAmt binary pic 9(5).*>     "126"
           05 Insurer pic x(39).        *>     "130"
           05 Details pic x(100).       *>     "169"
     02 Price comp pic 9(5)v99.         *>
"add(112,mpy(val(1,2,2),157))"
```

The offset of `Model` is straightforward, and is given by the formula "`36`." So is that of `Claims`, which is "`112`."

But because the array `Claim` can occur a variable number of times, the structure `History` is a variable-size field. Thus the offset of `Price`, which immediately follows `Claims`, requires a more complicated formula, involving the array stride (the distance between successive elements along a specific dimension). In the case when there is only one dimension for `Claim`, the formula for its stride is "`157`." Thus the formula offset of `Price` for a single dimension `Claim` is:

```
"add(112,mpy(val(1,2,2),157))"
```

The first argument of the val function is 1, meaning that the field containing the value at run-time, `NumClaims`, is in the same level-1 structure, `Used-Car`, as the field, `Price`, whose offset is specified by the formula. The other two arguments are 2 and 2. The first 2 refers to the second immediate subcomponent, `History`, of `Used-Car`. The second 2 means that the field to be dereferenced is the second component of `History`, that is, `NumClaims`.

In the case when `NumClaims` is greater than 1 (i.e., when `Claims` is a multi-dimension array) the offset for each element within `Claims` is 157 more than the offset for the previous dimension. For example, the offset formula for the second instance of `ClaimNo` is 112+157=269 while the third instance would be 269+157=426.

If the `OCCURS DEPENDING ON` object were in a separate structure, the third subcomponent of level-1 structure `Car-Data`, say, then the val function would be "`val(Car-Data,3)`."

2. COBOL structure mapping is top-down, although the direction doesn't make any difference unless the SYNCHRONIZED clause is specified on the data declaration. Specifying SYNCHRONIZED forces alignment of individual fields on their natural boundaries, and thus introduces "gaps" into the structure mapping. Consider the following data structure that is identical to the previous example, except for the SYNCHRONIZED clause:

```
*> Field                                  Offset
```

```
   01 Used-Car sync.                      *> "0"
     02 Summary.                          *> "0"
       03 Make pic x(36).                 *> "0"
       03 Model pic x(44).                *> "36"
       03 VIN pic x(13).                  *> "80"
       03 Color pic x(10).                *> "93"
         88 Red value 'Red'.
         88 White value 'White'.
         88 Blue value 'Blue'.
     02 History.                          *> "103"
       03 Mileage pic 9(6).               *> "103"
       03 NumClaims binary pic 9.         *> "110"
       03 InsCode pic x.                  *> "112"
       03 Claims.                         *> "113"
         04 Claim occurs 1 to 9 times
             depending on NumClaims.   *> stride(1) = "160"
                05 ClaimNo pic x(14). *> "113" plus one
slack byte after each instance of ClaimNo
                05 ClaimAmt binary pic 9(5).*> "128"
                05 Insurer pic x(39).       *> "132"
                05 Details pic x(100).      *> "171" plus
one slack byte after each instance of Details and one slack
byte after each instance of Claims
      02 Price comp pic 9(5)v99.        *>
"add(add(113,mpy(val(1,2,2),160)),3)"
```

To position the binary fields on their appropriate half-word or full-word storage boundaries, COBOL introduces padding, known as “slack bytes”, into the structure. Working top-down, this padding is introduced immediately before the field needing alignment. So there is one byte of padding between `Mileage` and `NumClaims`.

For an array, such as Claim, COBOL not only adjusts the padding within an element, but also the alignment of each element of the array. In the example, the first occurrence of Claim starts one byte past a full-word boundary. Because the field ClaimNo is three and a half words long, it ends three bytes past a full-word boundary, so COBOL inserts one byte of padding immediately before the binary full-word integer ClaimAmt. And to align subsequent occurrences, so that they too start one byte past a full-word boundary like the first, and can thus have an identical configuration, COBOL adds two bytes of padding at the end of each occurrence.

Finally, after padding, each occurrence of Claim (starts and) ends one byte past a full-word boundary, so COBOL puts three bytes of padding before the binary field Price. As a result of all these extra bytes, the formula for the offset of Price has changed considerably from the unaligned example, and is now:

```
"add(add(113,mpy(val(1,2,2),160)),3)"
```

There are several differences between the `OCCURS DEPENDING ON` construct and PL/I’s Refer option. Storage for COBOL structures is always allocated at the maximum size, whereas PL/I structures are allocated at the actual size specified by the Refer option. It is legal and usual to change the number of occurrences in a particular

instance of a variable-size COBOL array, and this has the effect of changing the location and offset of any fields that follow the array. For PL/I, the value of the Refer object of a particular instance of a structure is intended to be fixed during execution. Thus aligned objects following a variable-size field are always correctly aligned for each instance of the structure, because the amount of padding is computed uniquely for each instance, as determined by the Refer option. By contrast, the amount of padding for any aligned fields following a variable-size COBOL array is computed assuming the maximum array size, and is fixed at compile time. If the array is smaller than its maximum size, then the alignment will typically be incorrect. For instance in this example:

```
1 a sync.
      2 b binary pic 9.
      2 c pic x occurs 1 to 5 times depending on b.
      2 d binary pic 9(9).
```

COBOL inserts one byte between `c` and `d`. The alignment of `d` is therefore correct for only two values of `b`, the maximum, 5, and 2.

3. As noted above, the formulas describe not only offsets of fields within a structure, but also properties of arrays, such as bounds and strides. COBOL does not have true multi-dimensional arrays, although element references do use multiple subscripts. Instead, COBOL has arrays of arrays, as in the following simple example:

```
1 a. *< offset = "0"
   2 d1 occurs 5 times.    *< offset = "0"
                           *< lbound(1) = "1"
                           *< hbound(1) = "5"
                           *< stride(1) = "168"
      3 d2 occurs 6 times. *< offset = "0"
                           *< lbound(2) = "1"
                           *< hbound(2) = "6"
                           *< stride(2) = "28"
         4 el binary pic 9(9) occurs 7 times.  *< offset =
"0"
                           *< lbound(3) = "1"
                           *< hbound(3) = "7"
                      *< stride(3) = "4"
```

The program can refer to slices of the array by subscripting the higher-level container fields, for example, `d1(2)` or `d2(3, 4)`, but the normal kind of reference is to the low-level elements using the full sequence of subscripts, for instance, `el(4, 5, 6)`. To locate element `el(m, n, o)` using these stride formulas, one would take the address of `a` and add to it (m-1)*168 + (n-1)*28 + (o-1)*4. For COBOL, the lower bound of an array subscript is always 1. That is, the first element is always element(1), and vice versa.

Needless to say, any dimension of the array can have the `OCCURS DEPENDING ON` clause, and the array can be followed by other fields that complicates the formulas a lot. Consider the example:

```
1 a.
   2 x1 binary pic 9.          *< offset = "0"
   2 x2 binary pic 9.          *< offset = "2"
   2 x3 binary pic 9.          *< offset = "4"
   2 d1 occurs 1 to 5 times    *< offset = "6"
        depending on x1.       *< lbound(1) = "1"
                               *< hbound(1) = "val(1,1)"
                               *< stride(1) =
"mpy(val(1,2),mpy(val(1,3),4))"
      3 d2 occurs 1 to 6 times *< offset = "6"
         depending on x2.      *< lbound(2) = "1"
                               *< hbound(2) = "val(1,2)"
                              *< stride(2) = "mpy(val(1,3),4)"
         4 el binary pic 9(9)    *< offset = "6"
            occurs 1 to 7 times *< lbound(3) = "1"
          depending on x3.     *< hbound(3) = "val(1,3)"
                               *< stride(3) = "4"
    2 b binary pic 9(5).       *< offset = "see below!"
```

Computing the address of a particular element still involves the stride formulas, but these are no longer simple integers. The address of element `el(m, n, o)` in the above example is given by taking the address of `a` and adding to it:

```
(m-1)*stride(1) + (n-1)*stride(2) + (o-1)*stride(3), i.e.,

(m-1)*4*val(1,3)*val(1,2) + (n-1)*4*val(1,3) + (o-1)*4.
```

Similarly, these stride formulas are used in the formula for the offset of b:

```
"add(6,mpy(val(1,1),mpy(val(1,2),mpy(4,val(1,3)))))
```

7.3.8.2 PL/I

1. Given the following structure

```
dcl /* offset                                          */
        1 c unaligned              /*       "0"   */
          ,2 c1                    /*       "0"   */
            ,3 c2 fixed bin(31)     /*       "0"   */
            ,3 c3 fixed bin(31)     /*       "4"   */
          ,2 c4                    /*       "8"   */
            ,3 c5 fixed bin(31)     /*       "0"   */
            ,3 c6 fixed bin(31)     /*       "4"   */
            ,3 c7 fixed bin(31)     /*       "8"   */
          ,2 c8   fixed bin(31)     /*       "20"  */
          ,2 c9   char( * refer(c7) ) /*     "24"  */
          ,2 c10  char(6) /*   "add(24,val(1,2,3))" */
         ,2 c11   char(4)             /*
```

```
"add(add(24,val(1,2,3)),6)" */
                          ;
```

The offset of c3 would be given by the simple formula “4”, but the offset of c10 would be given by the formula:

```
"add(24,val(1,2,3))"
```

The first argument in the above val function is 1 that indicates the current structure, c. The subsequent arguments are 2 and 3, indicating that the third element, c7, of the second level-2 field, c4, is the field to be dereferenced.

The offset of c11 is equal to the offset of c10 plus the length of c10 and would be given by the following formula:

```
"add(add(24,val(1,2,3)),6)"
```

2. PL/I structure mapping is not top-down, and this can be illustrated by examining the mapping of the following structure:

```
dcl /* offset                              */
   1 a based,                          /* "0" */
      2 b,                             /* "0" */
       3 b1 fixed bin(15),       /*    "0" */
       3 b2 fixed bin(15),       /*    "2" */
       3 b3 fixed bin(31),       /*    "4" */
      2 c,                        /*
"add(8,mod(neg(val(1,1,1)),4))" */
       3 c1 char( n refer(b1)),  /*     "0" */
       3 c2 fixed bin(31);       /*     "val(1,1,1)" */
```

The value of b1 is given by val(1,1,1), and in order to put c2 on a 4-byte boundary, PL/I puts any needed padding before c (yes, not between c1 and c2), and hence the offset of c would be given by the following formula:

```
"add(8,mod(neg(val(1,1,1)),4))"
```

So if b1 contains the value 3, then this formula becomes add(8,mod(neg(3),4)), which evaluates to 9. I.e., there is one byte of padding between the structure b and the structure c.

3. The model also uses these formulas to specify the bounds and strides in an array, where the stride is defined as the distance between two successive elements in an array.

For example, in the following structure, the second dimension of a.e has a stride specified by the formula “4”, and the first dimension by the formula “20”:

```
dcl
   1 a,                                  /* offset = "0" */
      2 b(4) fixed bin(31),              /* offset = "0" */
                                         /* lbound(1) = "1" */
```

```
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "4" */
      2 c(4) fixed bin(31),         /* offset = "16" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "4" */
      2 d(4) char(7) varying,       /* offset = "32" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "9" */
      2 e(4,5) fixed bin(31);       /* offset = "68" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "20" */
                                    /* lbound(2) = "1" */
                                    /* hbound(2) = "5" */
                                    /* stride(1) = "4" */
```

This means that to locate the element a.e(m,n), one would take the address of a.e and add to it (m-1)*20 + (n-1)*4.

If the example were changed slightly to:

```
dcl
   1 a(4),                      /* offset = "0" */
                                /* lbound(1) = "1" */
                                /* hbound(1) = "4" */
                                /* stride(1) = "40" */
      2 b fixed bin(31),        /* offset = "0" */
      2 c fixed bin(31),        /* offset = "4" */
      2 d char(7) varying,      /* offset = "8" */
      2 e(5) fixed bin(31);     /* offset = "20" */
                                /* lbound(1) = "1" */
                               /* hbound(1) = "5"        */
                                /* stride(1) = "4"        */
```

then there is padding between d and e, but the user of the type descriptor can be blissfully unaware and simply use the stride and offset formulas to locate any given array element.

The stride for a is “40”, the stride for e is “4”, and the offset for e is “20.” This means that to locate the element a(m).e(n), one would take the address of a and add to it (m-1)*40 + 20 + (n-1)*4.

Finally, if the example were changed again to:

```
dcl
   1 a(4),                     /* offset = "0"           */
                               /* lbound(1) = "1"        */
                               /* hbound(1) = "4"        */
                               /* stride(1) = "40"       */
      2 b fixed bin(31),       /* offset = "0"           */
```

```
2 c(8) bit(4),          /* offset = "4"          */
                        /* lbound(1) = "1"       */
                        /* hbound(1) = "8"       */
                        /* stride(1) = "4"       */
2 d char(7) varying,    /* offset = "8"          */
2 e(5) fixed bin(31);   /* offset = "20"         */
                        /* lbound(1) = "1"       */
                        /* hbound(1) = "5"       */
                        /* stride(1) = "4"       */
```

then the computations for a.e are the same as above, but the computations for a.c become interesting.

The stride for a is still “40”, the stride for c is “4” (but this “4” is a count of bits, not bytes), and the byte offset for c is “4.” To locate the element a(m).c(n), one needs both a byte address and a bit offset. For the byte address, one would take the address of a and add to it (m-1)*40 + 4 + ((n-1)*4)/8. The bit offset of a(m).c(n) would be given by mod((n-1)*4,8).

7.3.9 Physical Representation Model: TDLang Interaction Diagram

The purpose of this section is to provide the reader with an overview of how the TDLang model is used to connect and integrate CAM, Type Descriptor, and interface models together. The Type Descriptor metamodel is a language-independent model used to convert a datatype into its expected language-specific type. This is accomplished by associating the base class, InstanceTDBase, to TDLangElement. As the parent class of all language model element classes, TDLangElement allows Type Descriptor to access the information regarding all language-specific data types for marshaling. Type Descriptor's association to the language elements via TDLangElement also provides the aggregate associations captured in the language models (i.e., the ComposedTypes associations for parent-child relationships). This ability to navigate up to parent or sibling elements is required to determine the value of various formula-based attributes in the Type Descriptor model. For example, in order for a child element C to determine it's offset formula value, it will need to navigate up to element B to find B's offset value and allocation size. The result of the adding element B's offset value and allocation size is element C's offset value.

Caching and navigation are two approaches to determining the parent value, but the navigation approach is superior to the cache approach in two respects. First, contents in the cache may become invalid as subscript values change from one child element to the next during runtime, resulting inaccurate cache data. Second, to fix this problem the marshaller will need to recalculate the values of each element at runtime, resulting in a decrease in performance. In the case when we apply navigation from the Type Descriptor model to the language models, we are able to quickly go from the child to the parent element to determine the formula information on a real-time basis. The navigation approach provides accurate values quickly without the need to perform recalculations.

The next diagram shows how language models associate to the Type Descriptor model via the TDLang model. Following the diagram is a brief explanation of what each class represents.

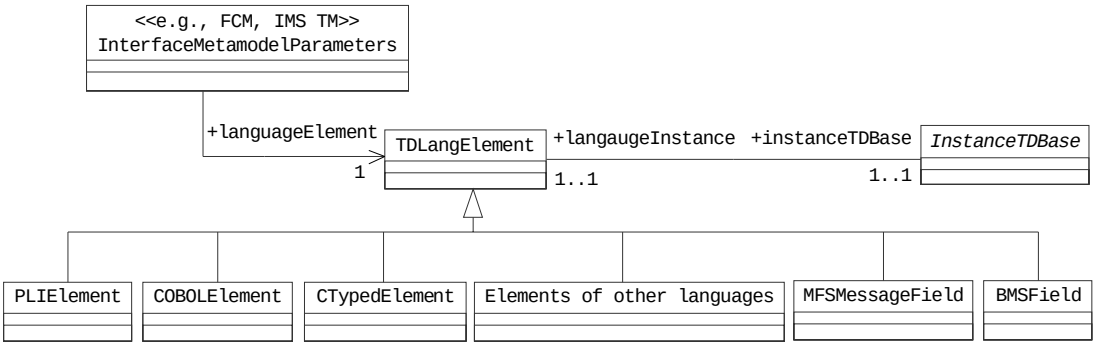


Figure 7-7 A View of Select CAM Models and Interface Models interacting with TDLang

7.3.10 Descriptions of TDLang Interaction Diagram

7.3.10.1 Interface Metamodel Parameters

Interface Metamodel Parameters represent a variety of input and output parameter classes which map to underlying language elements. Information on the language element's physical representation is captured by the Type Descriptor metamodel. Each instance of TDLangElement maps its corresponding physical representation in InstanceTDBase. TDLangElement navigates to InstanceTDBase via the instanceTDBase association. Examples of Enterprise Application Metamodel Parameters include ApplicationData (from IMS Transaction Message Metamodel), MFSMessageField (from IMS MFS Metamodel), and FCMParameter (from FCM Metamodel).

7.3.10.2 TDLangElement and Language Elements

As stated in Section 7.3.4.4 on page 7-8, TDLangElement is the parent class of all CAM language Element classes. Figure 7-7 on page 7-24 shows how any CAM language element can be modeled to support any given Interface Metamodel Parameter.

7.3.10.3 InstanceTDBase

As stated in Section 7.3.6.9, “InstanceTDBase,” on page 7-13, InstanceTDBase is used to represent the physical layout of each language element.

7.3.11 Sample Serialization of Convergent Metamodel

An example of how a marshaller might traverse the Type Descriptor-TDLang-Language model is as follows:

Given the following COBOL Data Declaration:

```
01     NAME.
     02     FIRST     PIC X(10).
     02     LAST      PIC X(10).
```

The following COBOL and Type Descriptor XMI instances would be serialized:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:COBOL="COBOL.xmi" xmlns:TypeDescriptor="TypeDescriptor.xmi">
  <COBOL:COBOLElement xmi:id="Element:NAME" name="NAME" level="01"
instanceTDBase="AggregateInstanceTD_1" sharedType="Type:NAME"/>
  <COBOL:COBOLComposedType xmi:id="Type:NAME">
    <element xmi:id="Element:NAME/FIRST" name="FIRST" level="02"
redefined="false" instanceTDBase="SimpleInstanceTD_1"
sharedType="Type:NAME/FIRST" initial="COBOLElementInitialValue_1"/>
    <element xmi:id="Element:NAME/LAST" name="LAST" level="02"
redefined="false" instanceTDBase="SimpleInstanceTD_2"
sharedType="Type:NAME/LAST" initial="COBOLElementInitialValue_2"/>
  </COBOL:COBOLComposedType>
  <TypeDescriptor:PlatformCompilerInfo xmi:id="PlatformCompilerInfo_1"
language="COBOL" defaultCodepage="8859_1" defaultBigEndian="false"
defaultFloatType="ieeeNonExtended" defaultExternalDecimalSign="ascii"
defaultAddressSize="mode32"/>
  <TypeDescriptor:AggregateInstanceTD xmi:id="AggregateInstanceTD_1"
offset="0" contentSize="20" size="20" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME"/>
  <COBOL:COBOLElementInitialValue xmi:id="COBOLElementInitialValue_1"
name="FIRST" initVal=" " valueKind="all_literal"/>
  <COBOL:COBOLAlphaNumericType xmi:id="Type:NAME/FIRST" usage="display"
pictureString="XXXXXXXXXX" synchronized="false" justifyRight="false"/>
  <TypeDescriptor:SimpleInstanceTD xmi:id="SimpleInstanceTD_1"
offset="0" contentSize="10" size="10" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME/FIRST" sharedType="StringTD_1"/>
  <TypeDescriptor:StringTD xmi:id="StringTD_1" addrUnit="word"
width="10" alignment="byte" lengthEncoding="fixedLength"
prefixLength="0" stringJustification="leftJustify" paddingCharacter=" "
characterSize="1"/>
  <COBOL:COBOLElementInitialValue xmi:id="COBOLElementInitialValue_2"
name="LAST" initVal=" " valueKind="all_literal"/>
  <COBOL:COBOLAlphaNumericType xmi:id="Type:NAME/LAST" usage="display"
pictureString="XXXXXXXXXX" synchronized="false" justifyRight="false"/>
  <TypeDescriptor:SimpleInstanceTD xmi:id="SimpleInstanceTD_2"
offset="10" contentSize="10" size="10" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME/LAST" sharedType="StringTD_2"/>
  <TypeDescriptor:StringTD xmi:id="StringTD_2" addrUnit="word"
width="10" alignment="byte" lengthEncoding="fixedLength"
prefixLength="0" stringJustification="leftJustify" paddingCharacter=" "
characterSize="1"/>
</xmi:XMI>
```

Of particular interest is how the offsetFormula is determined. In order to determine the offsetFormula value of element LAST, the model needs to be able to navigate upward from LAST's SimpleInstanceTD to FIRST's SimpleInstanceTD to determine the offsetFormula and allocSizeFormula attributes of FIRST. Formula-based values can either be static (serialized during import time) or dynamic (serialized during runtime). It is this capability to navigate back-and-forth from language models to Type Descriptor that allows us to determine how to marshal each language element.

Formula-based attributes in the Type Descriptor model are typed as String in order to support both calculation and numeric values. Runtime determined values such as COBOL's Occurs-Depending-On clause will have calculation formulas as its value (e.g., "20+10x") while static values will use numeric values (e.g., allocSizeFormula of FIRST is "10"). Calculation formulas will be evaluated by a "Formula Evaluator", which takes the formula String as input and returns the calculated numeric value when runtime information is available (e.g., once the 'x' value of formula "20+10x" is determined we can return a numeric value). In the case of an numeric value (evaluated integer), simply pass the attribute value into a "Formula Evaluator" program and the integer representation of the string will be returned. The formulas in the Type Descriptor model should be generic for all languages. Therefore, the "Formula Evaluator" will cover all languages (COBOL, C, C++, PL/l, etc.).

Part 3 - Profile Definition

Contents

This section contains the following chapters.

Section	**Page**
8. Collaboration Modeling	8-1
9. Activity Modeling	9-1

The profile presented here focuses on two main modeling approaches, based on collaborations and based on activities. These are described in Chapters 8 and 9, respectively.

The collaboration-modeling approach is based on a modeling framework of classes that provide detailed definitions of the semantics of the collaboration. It is thus useful for providing the detailed specification of message flows in the design of integration subsystems.

The activity-modeling approach is based on the use of activity graphs. This approach is particularly useful for showing the overall control and data flow required for integration, typically at a higher level than in collaboration modeling.

Casting the metamodel as a UML profile allows EAI architecture models to be notated using standard UML notation. This means that most UML tools (specifically ones which support the extension mechanisms of UML, such as stereotypes and tagged values) can be used to define EAI architecture models.

Standard practice for defining UML profiles has been adopted. A mapping of metamodel classes to their base UML classes, with accompanying stereotypes, tagged values and constraints is summarised for each approach. An implementation of this mapping can be used, for example, to generate metadata conforming to the EAI

metamodel from XMI generated from models notated using the UML profile. Specialized EAI tools will more likely use the metamodel than the UML profile as a basis for storing and manipulating models.

The art of defining a UML profile is to provide the best fit possible with UML, so that the notation is natural for a modeler in the relevant domain (EAI in this case), and fits with one's general intuitions about the the meaning of the elements of UML that are used in the profile. The profile described here has been designed with these principles in mind.

Collaboration Modeling 8

8.1 Overview

8.1.1 General Approach

The Collaboration Profile for EAI makes use of UML class and collaboration diagrams to notate EAI models. The main parts of the profile are:

- Notation for terminals
- Notation for operators
- Notation for resources
- Notation for message formats

Operators are notated by class diagrams, which declare the input and output terminals of the operator and the message formats of those terminals. The class diagram can also be annotated with the definition of the operations performed when manipulating incoming messages to generate outgoing messages.

For compound operators, class diagrams also specify the component operators of the compound, which may, themselves, be compound operators. Collaboration diagrams are used to show how its components are connected together.

Different kinds of terminals are defined by appropriate stereotypes on UML Class. Specific, named terminals are identified with operators via associations.

Different kinds of operator are identified by appropriate stereotypes on UML Class.

Some operators make use of resources. Resources are notated by classes, with stereotypes used to capture the different kinds of format.

Message formats are notated by classes, with stereotypes used to capture the different kinds of format.

8.1.2 Use of UML operations

There are places where UML operations have been used with specific names to 'carry' certain pieces of metadata within a model defined by the profile. For example, when one defines a terminal, it is necessary to define an operation called handle whose return type determines the format of message content that the terminal can handle; when one defines a filter, it is necessary to define a boolean operation allow which determines, for a message supplied as argument, the conditions under which a message can pass through the filter. This approach to encoding this information was taken, because it accords with one's intuitions about the meaning of UML and of UML operations in particular. For example, one is able to explain what a filter does by referring to its allow operation - only incoming messages for which the allow operation evaluates to true get passed on.

It should be stressed that the operations themselves imply nothing about the scheme used to implement models, though clearly the information they hold will need to be carried through in some way. Indeed, most implementations are likely to work from the metamodel direct (as this issue does not arise there) and the profile just used as a means of defining models using UML notation, which can then get converted to instances of the metamodel for subsequent processing.

Any definition of an operation used in operator specifications must be provided as part of the specificatoin of that operation. There are many ways to show the definition of UML operations, which will depend on specific organizational practices and/or support provided by UML CASE tools. If tools do not support the display of operation specifications on diagrams (as many don’t) a UML note may be used in addition to repeat the diefinition on the diagram. In this document, the specification of operations in examples is relayed by notes on the diagrams.

8.1.3 Concrete Notation

Only raw stereotypes have been defined in this profile. The user may replace these with concrete icons at his or her discretion.

8.1.4 Chapter structure

The remainder of this chapter provides a detailed description of each of the four parts of the profile. Each part is described stereotype by stereotype, using generic examples for illustration. The constraints that apply in the context of a particular stereotype are also defined. The detailed descriptions are followed by a section describing the mapping of the EAI metamodel to the elements of the profile. This section also provides a summary of the stereotypes used in the profile, and follows the format laid down by UML 1.4.

8.2 Terminals

The terminals of an operator are shown by associations to classes with stereotypes <<input>> (for input terminals) and <<output>> (for output terminals), from classes with operator stereotypes (see sections below). Figure 8-1 gives a prototypical

example, showing the definition of terminals for a primitive operator . As shown, the primitive operator has two input terminals, names “in” and “queueIn”. While both these terminals handle the same kind of message format, the latter is specifically known to be a queued terminal. The primitive operator is also shown to have two output terminals, named “out1” and “out2”.

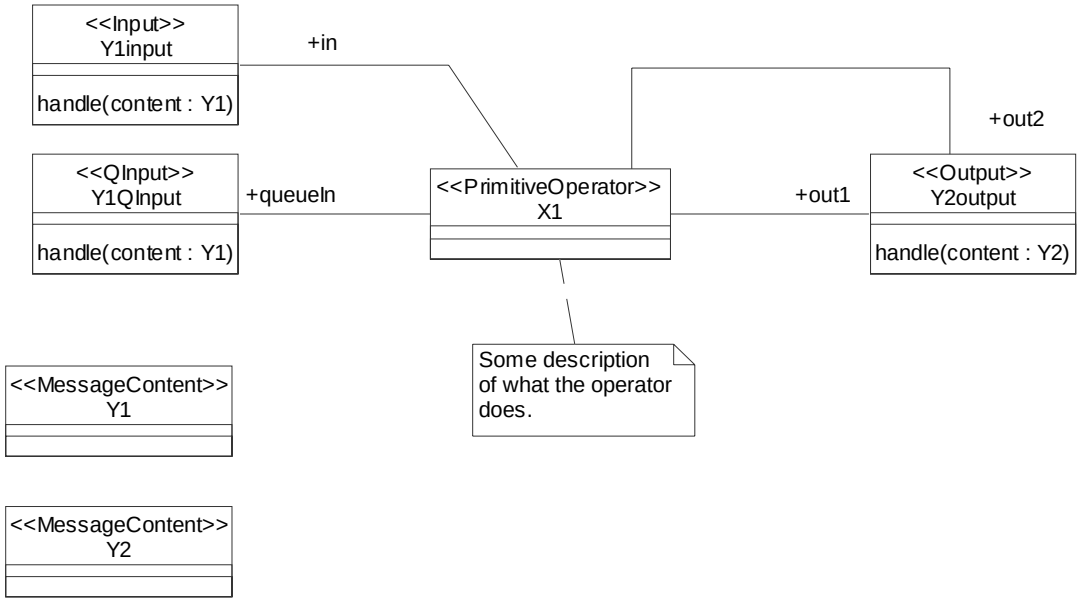


Figure 8-1 Class diagram for prototypical primitive operator with terminals

An input terminal is responsible for conveying incoming messages to the operator, while an output terminal is responsible for conveying outgoing messages away from the operator. The names of the terminals with respect to the operator are specified as labels on the appropriate association end. The associations are navigable only from operator to terminal, and they have cardinality 1. These markings (which may never change) may be omitted from the diagram (tool permitting) to avoid clutter. Any other properties and inconsistent with the profile. In general, operators may have one or more input and one or more output terminals. The number and names of the input and output terminals may be constrained for specialist primitive and compound operators.

Terminals can handle messages with a specified content format. This is indicated by declaring an operation handle on the class defining terminal kinds (i.e., classes with stereotypes <<input>> and <<output>>) which takes one argument of the specified format. Formats are specified by classes with a stereotype <<LangElement>> or one of its substereotypes, or stereotype <<MessageContent>> or one of its substereotypes. For most operators (adapters are the exceptions), the stereotype will usually be <<MessageContent>> corresponding to the generic format for message content.

It is not the role of this specification to say how a terminal handles its messages. However, the stereotypes <<QInput>> and <<QOutput>> may be used to indicate that handling is performed using a queue. Unless stated otherwise (e.g., as a constraint), it is assumed that terminals defined for any kind of operator may be plain or queued.

Finally, dynamic connection of terminals is supported. That is, it is possible to send some operators (for example routers) a message containing a terminal identifier, so that the operator can add or remove that terminal from the list of targets of one or more of its output terminals. The targets of an output terminal are the terminals connected to it.

Constraints

There should only be one input and output class per handle format/stereotype pairing, and the name of this class will be a concatenation of the format name and the stereotype name.

The type of the content parameter of the handle operation must have a stereotype of <<LangElement>> or one of its sub-stereotypes, or of <<MessageContent>> or one of its substereotypes.

8.3 Operators

8.3.1 Primitive Operator

Figure 8-1 on page 8-3 also shows a prototypical example of the definition of a primitive operator.

Primitive operators are useful for notating operators which have no internal structure (or whose internal structure is of no interest) such as system applications. A generic primitive operator is shown as a class with a stereotype <<PrimitiveOperator>>. The class may have an associated note (corresponding to EAIAnnotation in the metamodel) for recording a description of what the operator does.

Constraints

The type of content of the terminals of a generic primitive operator must have a stereotype <<MessageContent>> or one of its substereotypes.

8.3.2 Transformers and Database Transformers

Figure 8-2 shows the general format of the notation used to define a transformer, which is represented by a class with stereotype <<Transformer>>. A transformer uses the transform operation to transform the content of the input message and then sends the transformed message via the single output terminal of the transformer.

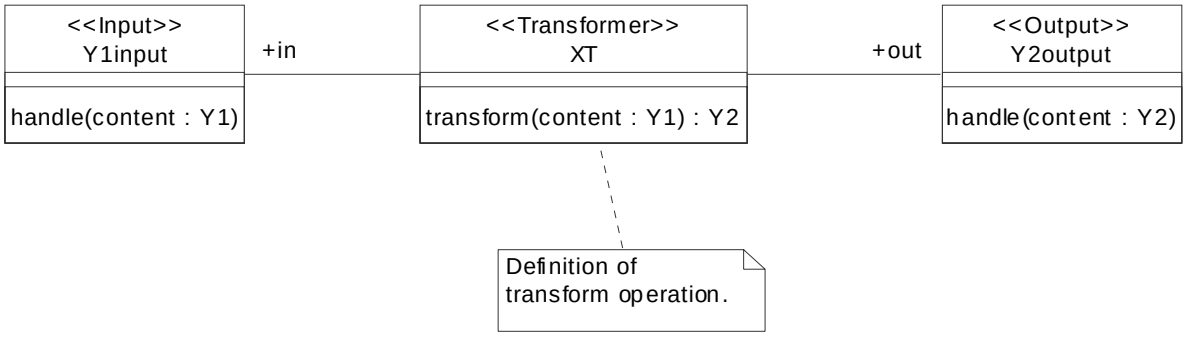


Figure 8-2 Class diagram for prototypical transformer

A database transformer is just like a transformer, except that it accesses a database in order to perform the transform operation. In this case, the stereotype <<DBTransformer>> is used, and this requires a database resource to be declared, as in Figure 8-3.

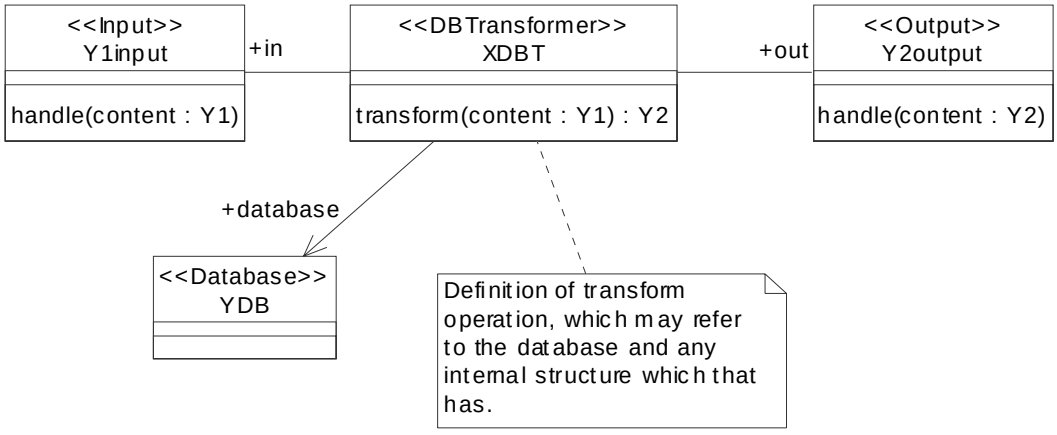


Figure 8-3 Class diagram for prototypical database transformers

Additionally, the definition of transform may make reference to this attribute.

Constraints

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the transform operation.

The type of content of the terminals of a transformer must have a stereotype <<MessageContent>> or one of its substereotypes.

For database transformers, there must be a directed association to a database resource (i.e., a class with stereotype <<Database>>) with the rolename “database” at the database resource end.

8.3.3 Filters

Figure 8-4 shows the general format of the notation used to define a filter.

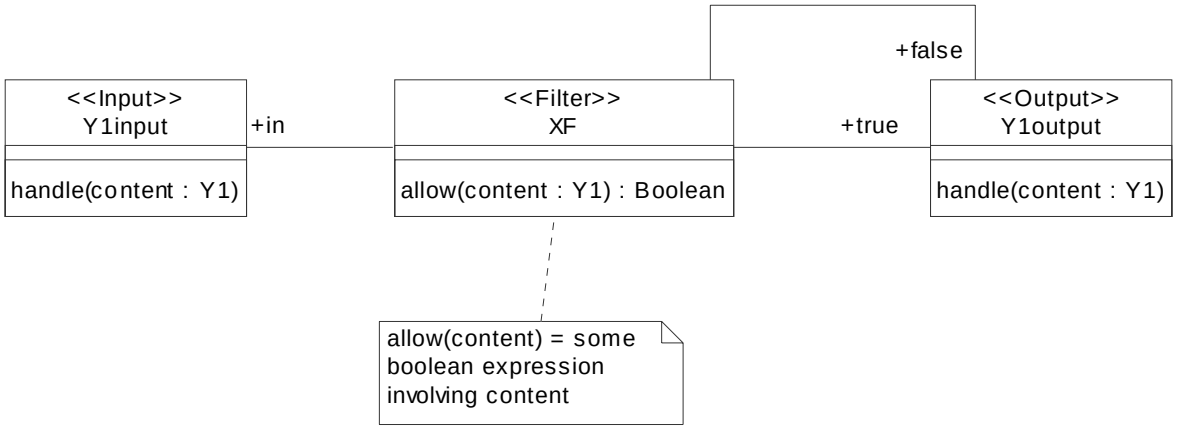


Figure 8-4 Class diagram for prototypical filter

A filter does not modify the content of the messages it receives. However, a filter only passes on those messages whose content meets specific criteria. When a filter is triggered, it uses the allow operation to test if the content of the input message meets the criteria. If so, the content is sent to the true output terminal, otherwise it is sent to the false terminal.

Constraints

The content format of the input and output terminals must match that of the parameter of the allow operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

8.3.4 Streams

For operators described so far it is assumed that messages are always received in the order that they are sent and that there is basically no delay in their transmission. In reality, there are some cases where a stream of messages may be received in a different order than that in which they are sent and they may be received at a different rate than that at which they are sent. A stream operator is used to model this. Figure 8-5 shows the general format of the notation used to define a stream operator.

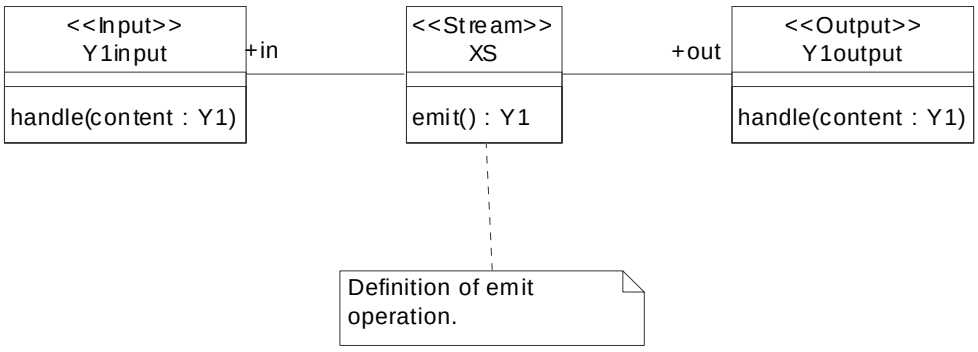


Figure 8-5 Class diagram for prototypical stream

Messages that arrive from the input terminal do not get passed on, but instead are stored in a buffer or some other appropriate data structure. The emit operation defines the algorithm used to decide when and in what order messages get emitted to the output terminal. Abstractly, one can imagine a loop that continually calls the emit operation. It returns a message to be put on the output terminal at each call. There may be a delay between its being called and its returning a message.

Constraints

The content format of the terminals must match that of the result of the emit operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

8.3.5 Post Daters

Figure 8-6 shows the general format of the notation used to define a post dater.

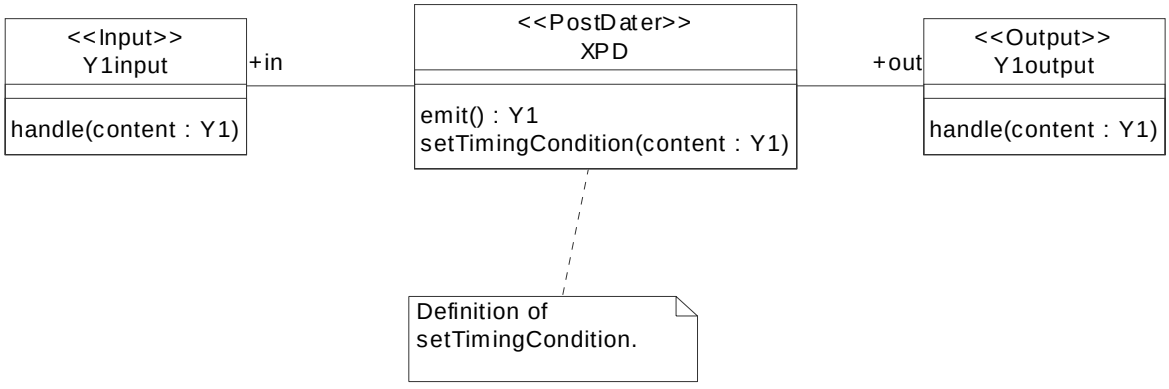


Figure 8-6 Class diagram for prototypical post dater

A post dater is specified using the <<PostDater>> stereotype. A special kind of stream is a post dater. On receipt of a message at its input terminal, it adds the message to the buffer, and creates an individual timingCondition for it. The timingCondition is derived from the content of the input message by the setTimingCondition operation. A post dater holds the message until its individual timing condition is met and then emits it from its out terminal.

As the definition for emit is fixed for post daters, only a definition for setTimingCondition should be provided.

Constraints

The content format of the terminals must match that of the result of the emit operation and the parameter of the setTimingCondition operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

8.3.6 Aggregators

Figure 8-7 shows the general format of the notation used to define an aggregator.

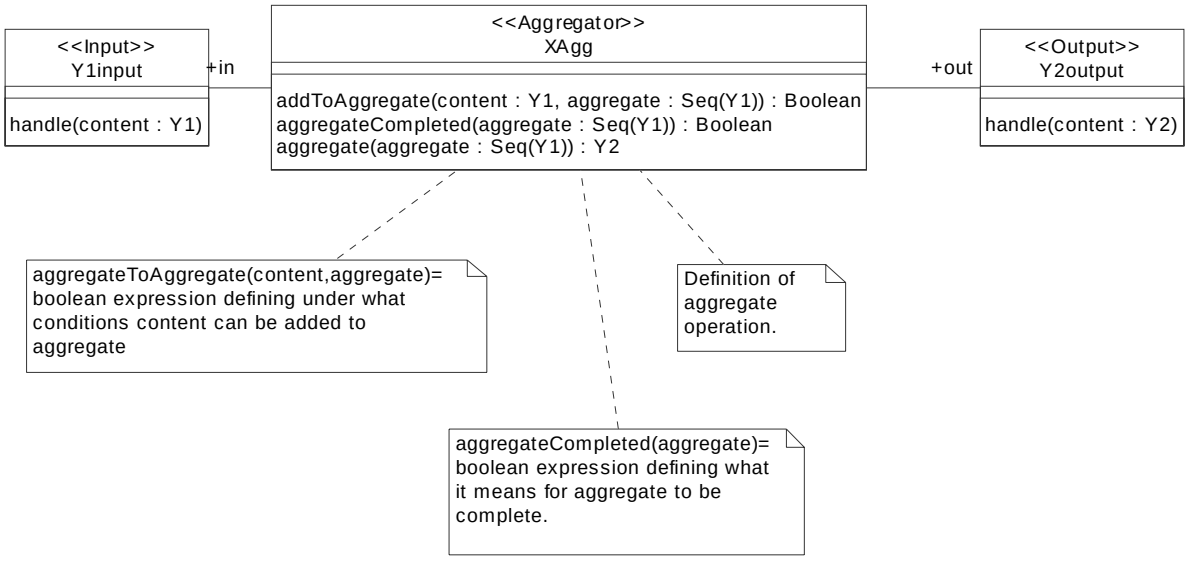


Figure 8-7 Class diagram for prototypical aggregator

An aggregator operator is indicated by the <<Aggregator>> stereotype. It creates aggregate messages based on one or more message aggregation specification, each of which is modeled by an associated class with the <<MessageAggregation>> stereotype. (Note that an aggregator can create multiple aggregates either by having an association with a multiplicity of greater than one with the same message aggregation class, in which case all aggregates share the same specification, or by having multiple associations with different message aggregation classes.)

On receipt of a message at its input terminal, the aggregator operator adds the message to each aggregate for which the addToAggregate condition (which will depend on the message header or body contents) evaluates to true.

Each time a message is added to an aggregate, the aggregateComplete condition is evaluated for that aggregate. If it evaluates to true, then a message is constructed from the messages it holds and is sent on the output terminal. The mapping from the messages contained in the aggregate to the message sent is specified by the aggregate operation.

If no aggregateComplete evaluates to true, then no message is sent.

Constraints

The type of content of the terminals must have a stereotype <<MessageContent>> or one of its substereotypes.

The aggregator class must have associations with one or more classes with the stereotype <<MessageAggregation>>.

A class stereotyped <<MessageAggregation>> must have addToAggregate, aggregationComplete and aggregate operations.

The addToAggregate operation of each message aggregation class must have two arguments, the first of which matches the content format of the in terminal of the aggregator operator and the second of which is a sequence of this content format, and a result of type Boolean.

The aggregationComplete operation of each message aggregation class must have a single argument whose type is a sequence of the message content format of the in terminal of the aggregator operator and a result of type Boolean.

The aggregate operation of each message aggregation class must have a single argument whose type is a sequence of the message content format of the in terminal of the aggregator operator and a result whose type matches the content format of the out terminal of the aggregator operator.

8.3.7 Timers

Figure 8-8 shows the general format of the notation used to define a timer.

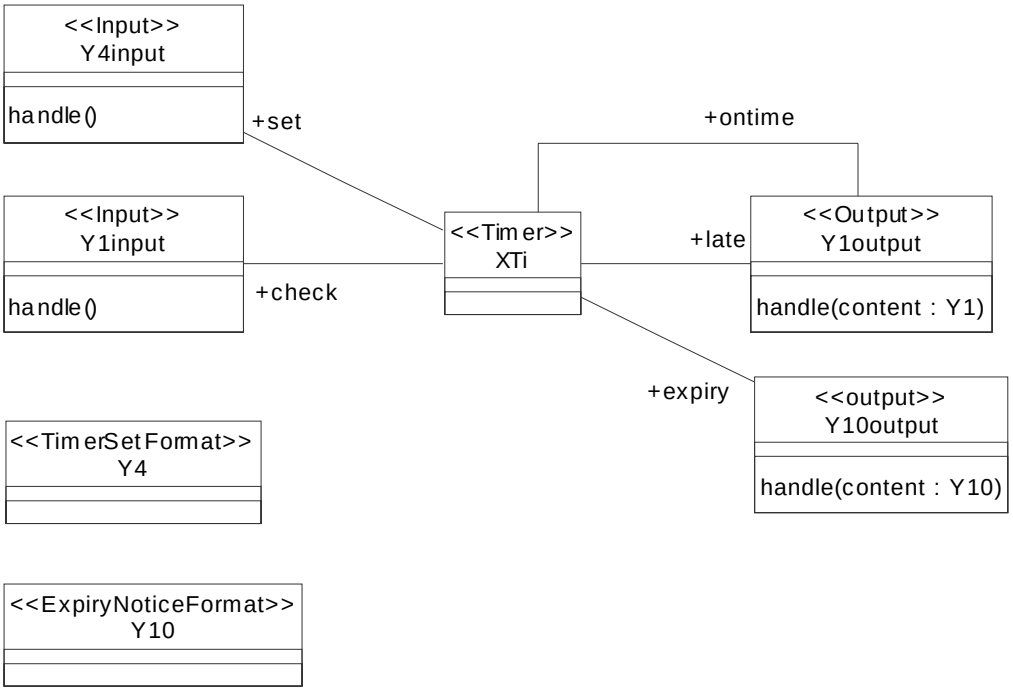


Figure 8-8 Class diagram for prototypical timer

A timer is specified using the <<Timer>> stereotype. It processes a message on its set terminal that specifies a timer set message which contains a pair comprising a timer and a correlation condition. This gets added to the timer's list of condition pairs. When a timer receives a message from the check terminal, it looks through its list of condition pairs and sees if the message satisfies any of the correlation conditions. If so, then the timer condition is examined to see if it has been met, and, if so, the message is past onto the ontime terminal. Otherwise it is passed onto the late terminal. If it does not meet any correlation condition, it is assumed the message is on time and therefore passed onto the ontime terminal.

Whenever a timer condition from the list of condition pairs expires, an expiry notice is sent to the expiry terminal.

Constraints

The input terminals must be labelled set and check. The output terminals must be labelled ontime, late and expiry.

The content format of the check, late and ontime terminals must be the same. This type must have stereotype <<MessageContent>> or one of its substereotypes.

The type of content of the set terminal must have a stereotype <<TimerSetFormat>>.

The type of content of the expiry terminal must have a stereotype <<ExpiryNoticeFormat>>.

8.3.8 Routers

Figure 8-9 shows the general format of the notation used to define a router.

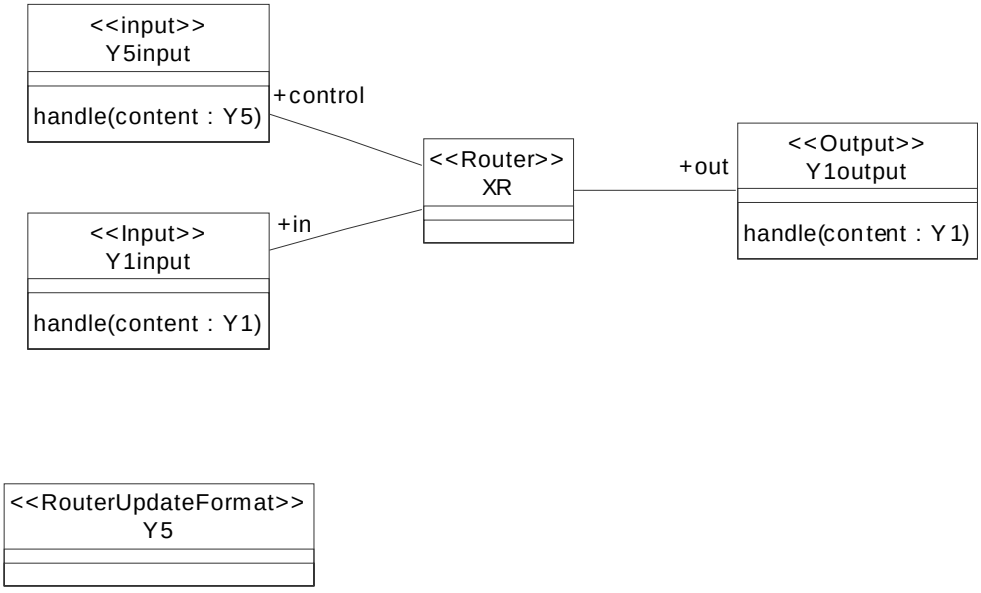


Figure 8-9 Class diagram for prototypical router

Figure 8-16 shows the general format of the notation used to define a router.

A router is specified using the <<Router>> stereotype. When a router receives a message on its “in” terminal, it resends a copy, via its out terminal, to all terminals listed in an associated routing table. The routing table is shown as a class with stereotype <<RoutingTable>>, with a directed association from the router to it, with role name “routingTable”.

A router updater can be used to make dynamic additions or removals of target terminals to or from a routing table. This can be used to model a simple publication channel for messages. A router updater is specified using the <<RouterUpdate>> stereotype, with a directed “routerUpdater” association from the router updater to a routing table. When a router updater receives a message on its “control” terminal that is in a router-update format, it performs the adds or removes given in that message on the associated routing table.

Note that, if a router has static EAILinks on its “out” terminal, then the target input terminals linked to it by those EAILinks are automatically added as the initial contents of the routing table for the router. If no dynamic updating is to be done on this initial

contents (that is, no router updater will ever act on it), then it is not necessary to show the routing table explicitly in the model, and the router need not have a routingTable association.

Constraints

A router must have a single input terminal labeled “in” and a single output terminal labeled “out”. The type of content of the terminals of a router must be stereotyped by <<MessageContent>> or one of its substereotypes.

A router updater must have a single input terminal labeled “control” and no output terminals. The type of content of the “control” terminal of a router updater must have the stereotype <<RouterUpdateFormat>>.

A router updater must have a directed association to a class sterotyped <<RoutingTable>>, with the role name routingTable.

8.3.9 Subscription Operators

Figure 8-10 shows the general format of the notation used to define a subscription operator.

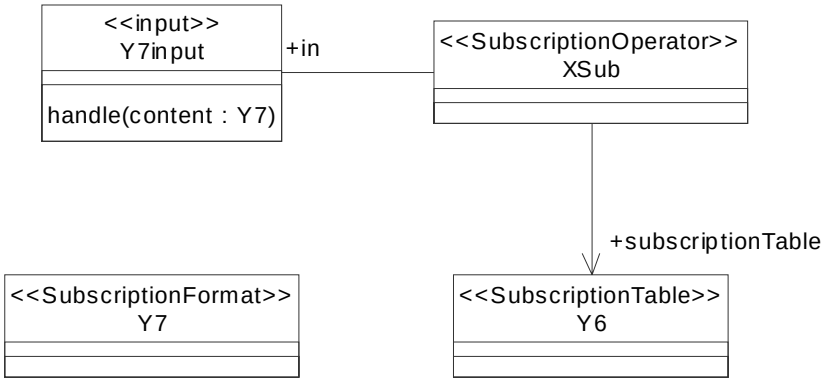


Figure 8-10 Class diagram for prototypical subscription operator

A subscription operator is specified using the stereotype <<SubscriptionOperator>>. It expects a message of subscription format as input. This carries a subscription comprising a terminal identifier and a filter definition. When it receives one of these messages, it adds the subscription to its subscription table. A subscription message may also request subscriptions for a terminal to be canceled.

Constraints

The type of content of the input terminal must have a stereotype <<SubscriptionFormat>>.

There must be a directed association to a subscription table (i.e., a class with stereotype <<SubscriptionTable>>). This should be labeled subscriptionTable.

8.3.10 Publication Operators

Figure 8-11 shows the general format of the notation used to define a publication operator.

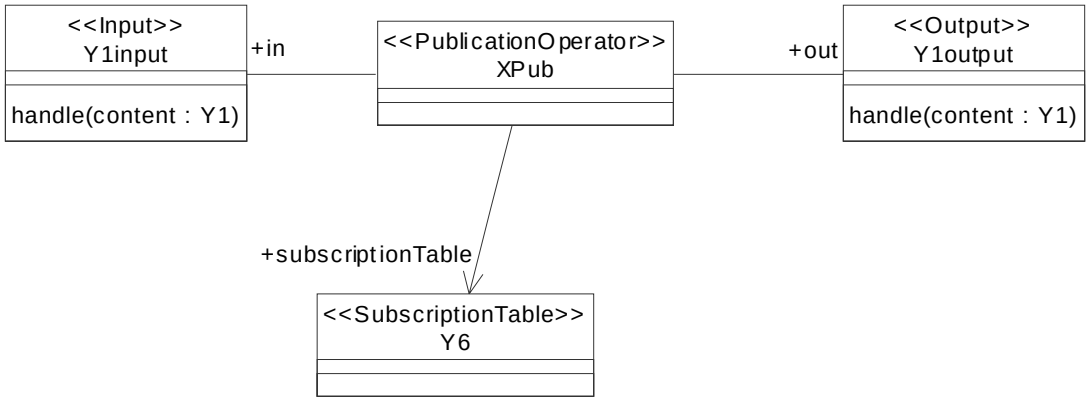


Figure 8-11 Class diagram for prototypical publication operator

A publication operator is specified using the stereotype <<PublicationOperator>>. Messages sent to the input terminal are sent from the output terminal to each subscriber (terminal) if the message passes the filter specified by the subscription for that subscriber.

A publication operator is accompanied by at least one subscription operator when defined as part of an architecture. See Section 8.3.12.5, “Publish and Subscribe,” on page 8-21 for details.

Constraints

The type of content of both terminals must be the same and have a stereotype <<MessageContent>> or one of its substereotypes.

There must be a directed association to a subscription table (i.e., a class with stereotype <<SubscriptionTable>>). This should be labeled subscriptionTable.

8.3.11 Topic Publishers

Figure 8-12 shows the general format of the notation used to define a topic publisher.

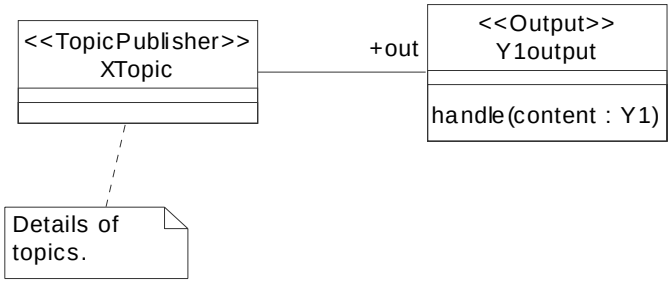


Figure 8-12 Class diagram for prototypical topic publisher

A topic publisher is specified using the stereotype <<TopicPublisher>>. It is kind of source, which sends only sends messages to the output terminal on a set of specified topics. Details about the topics may be added as a note. The content type of the output terminal may also be an indicator of the kinds of topics published on.

Topic publishers are usually connected to the input terminal of a publication operator. See Section 8.3.12, “Compound Operators,” on page 8-14 for details.

Constraints

There is a single output terminal.

The type of content of the output terminal must have a stereotype <<MessageContent>> or one of its substereotypes.

8.3.12 Compound Operators

Compound operators allow more complex message transformation and routing behavior from a (possibly nested) composition of individual operators to be modeled. Indeed any non-trivial architecture will be modeled as a compound operator whose components will be primitive or other compound operators.

Compound operators are defined using a combination of class and collaboration diagrams.

8.3.12.1 Class diagrams

Figure 8-20 shows the class diagram for an example compound operator, which is specified using the stereotype <<CompoundOperator>>. The example is taken from Chapter 10.

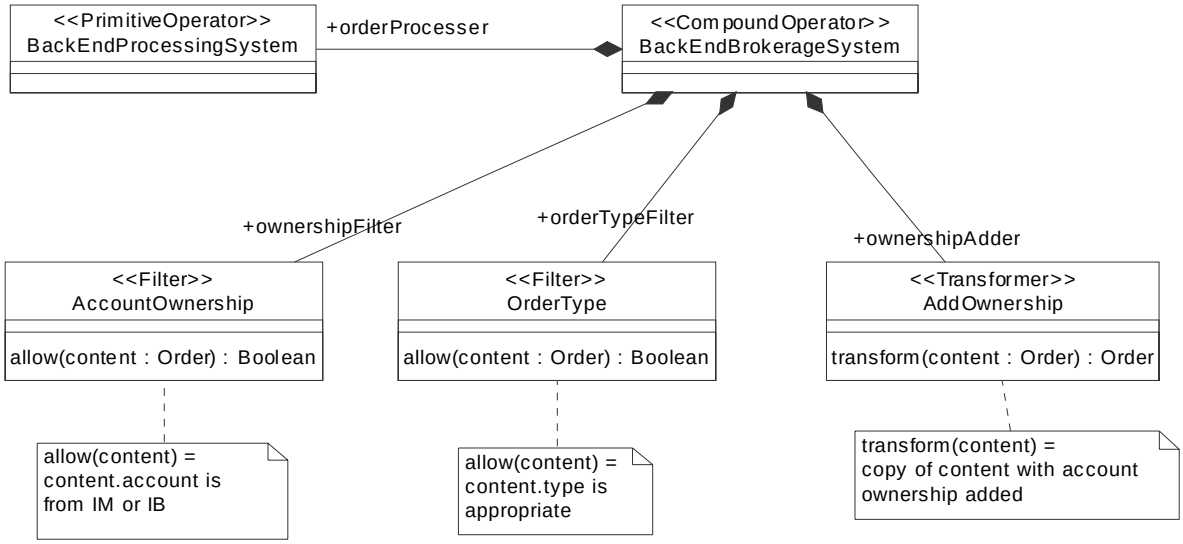


Figure 8-20 Class diagram for example compound operator

This defines a compound operator called BackEndBrokerageSystem with three components: two filters and a transformer. The primitive operator, filters and transformers are defined as previously discussed. Components are shown by means of a composite association targeted on a class representing an operator definition. Although the components shown here are all primitive operators, they may be compound operators, as illustrated by Figure 8-21.

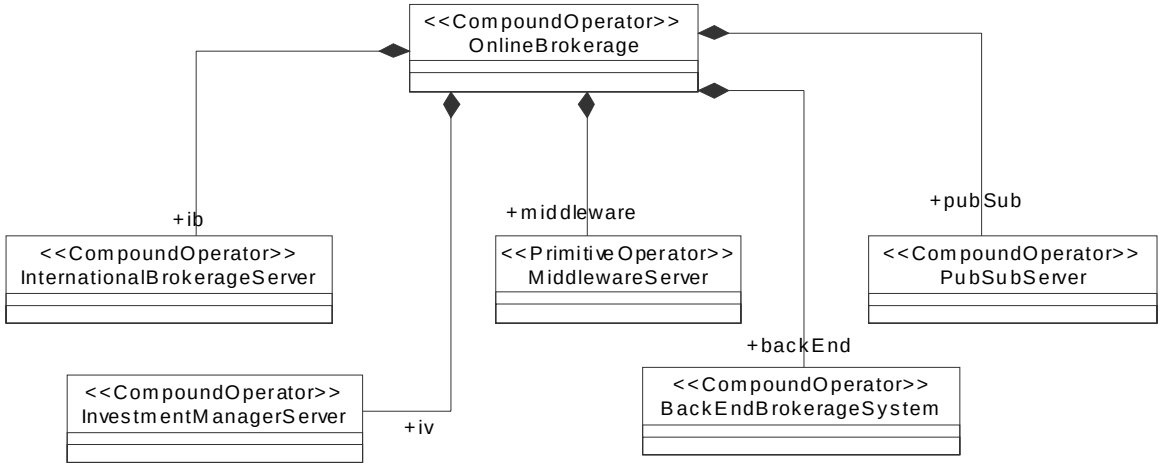


Figure 8-21 Class diagram for a compound operator with compound components

Note, in this diagram, that one component of an OnlineBrokerage is a BackEndBrokerageSystem, which, as we have already seen, is a compound operator.

As with primitive operators, class diagrams can also be used to define the terminals of a compound operator. The terminals of BackEndBrokerageSystem are defined by Figure 8-22.

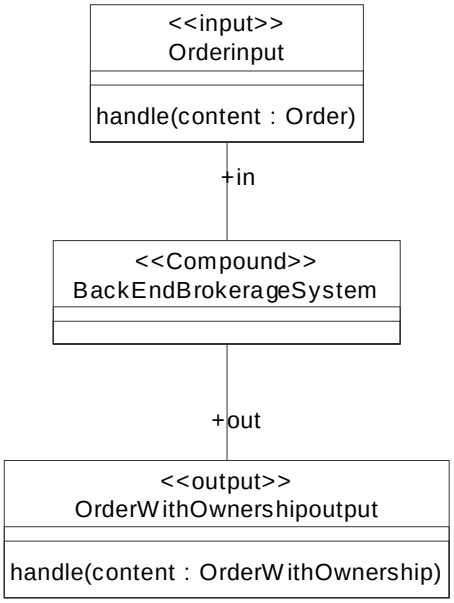


Figure 8-22 Terminals for example of compound operator

Figure 8-22 does not show the connectivity of the components, that is, how the terminals of the components are connected together and connected to the terminals of the compound operator. A collaboration diagram is used to show the connectivity of the components.

8.3.12.2 Collaboration Diagrams

The collaboration diagram corresponding to Figure 8-20 is given in Figure 8-23.

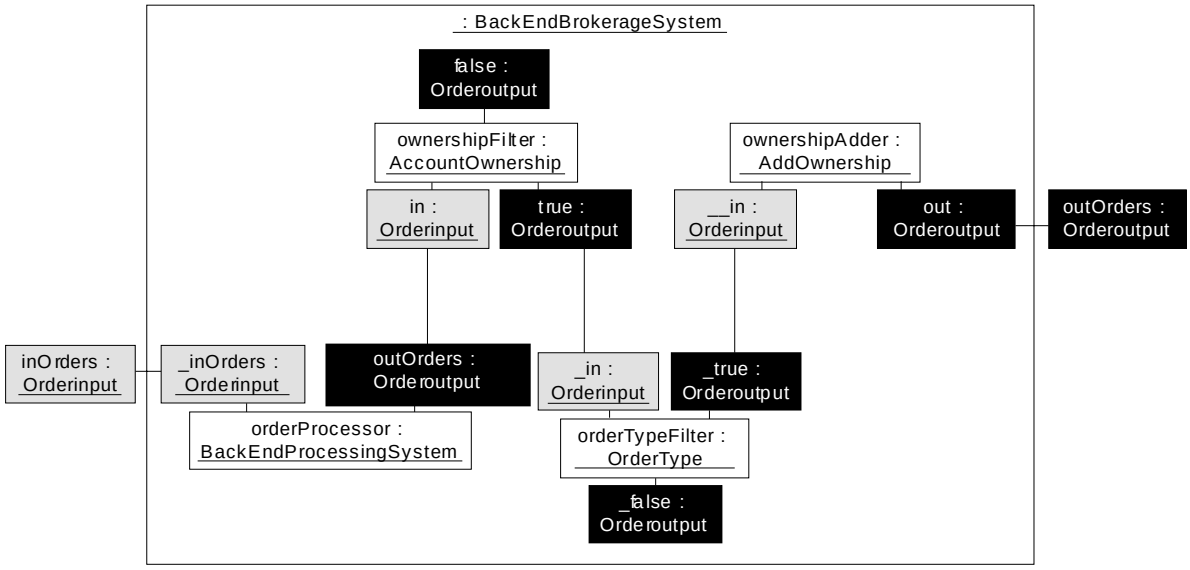


Figure 8-23 Collaboration diagram for example compound operator

This shows:

- The components of the compound as objects contained in an object representing the compound
- The terminals of the components (also contained in the compound), and the terminals of the compound itself (outside the compound).

The names of the objects correspond to the names of the components or terminals, as declared on the class diagram. The compound object has no name, as it represents an arbitrary operator of the compound-operator type being defined. We have used gray (or black) to distinguish input (or output) terminals from operators; this is just a convention. Connection of components is shown by connecting the terminals in an appropriate way (see Section 8.3.12.6, “Constraints,” on page 8-22 for a definition of what is appropriate)[1]. Ownership of terminals by an operator is also shown through links; the convention is to cluster terminals around their operator.

The UML 1.4 metamodel requires links to be connected to associations. In this case, the associations are redundant, but any tool strictly conforming to UML 1.4 should force the link to be associated with an association. To accommodate this, all UML for

1. Underscores on names are used to ensure uniqueness, a requirement of the tool used.

EAI models should include a class EAITerminal, from which all Terminal classes inherit. It has an association to itself with cardinality 0...* on each end, and its end names are left empty. All terminal-to-terminal links are instances of this association.

Sometimes one may wish to be explicit about whether the connection between terminals is synchronous or asynchronous. This is shown by putting a message on the link, which is marked as asynchronous or synchronous. Figure 8-24 shows the standard UML notation for this.

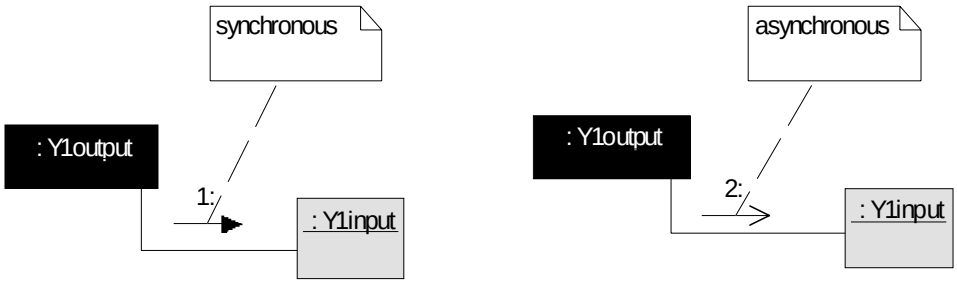


Figure 8-24 Synchronous and asynchronous links

The arrow of the message goes in the direction of the message flow (output to input when terminals of components of a compound are connected).

8.3.12.3 Components of the same type

A situation that the modeler should be aware of is the case where a compound may include two components of the same type of operator. This is illustrated by Figure 8-25 and Figure 8-26. The point to note is that there are two components of StandardIBSystem operator type (which is evident from the two associations to the StandardIBSystem class on the class diagram) and two objects of this class on the collaboration diagram.

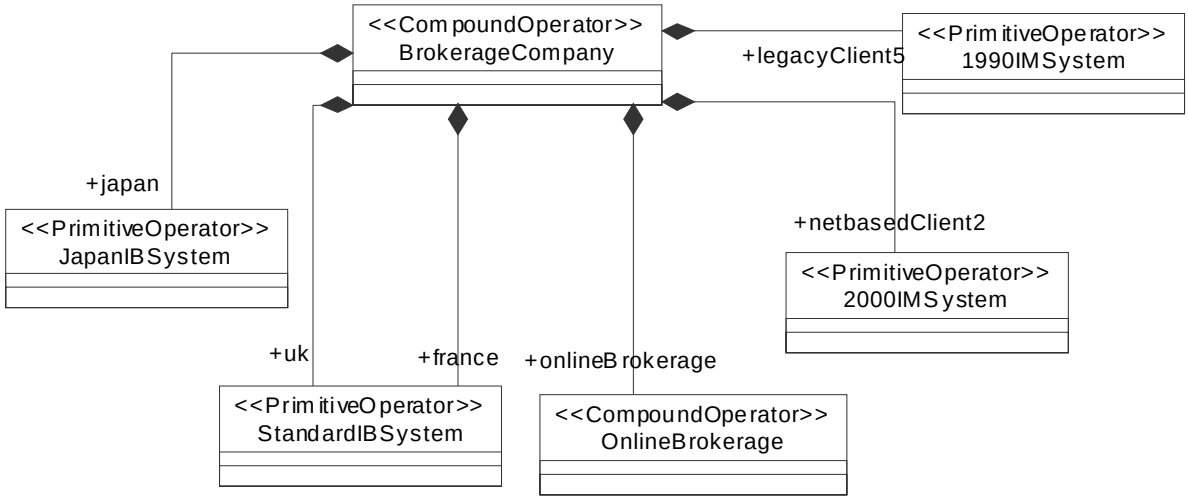


Figure 8-25 Class diagram for example with components of same type

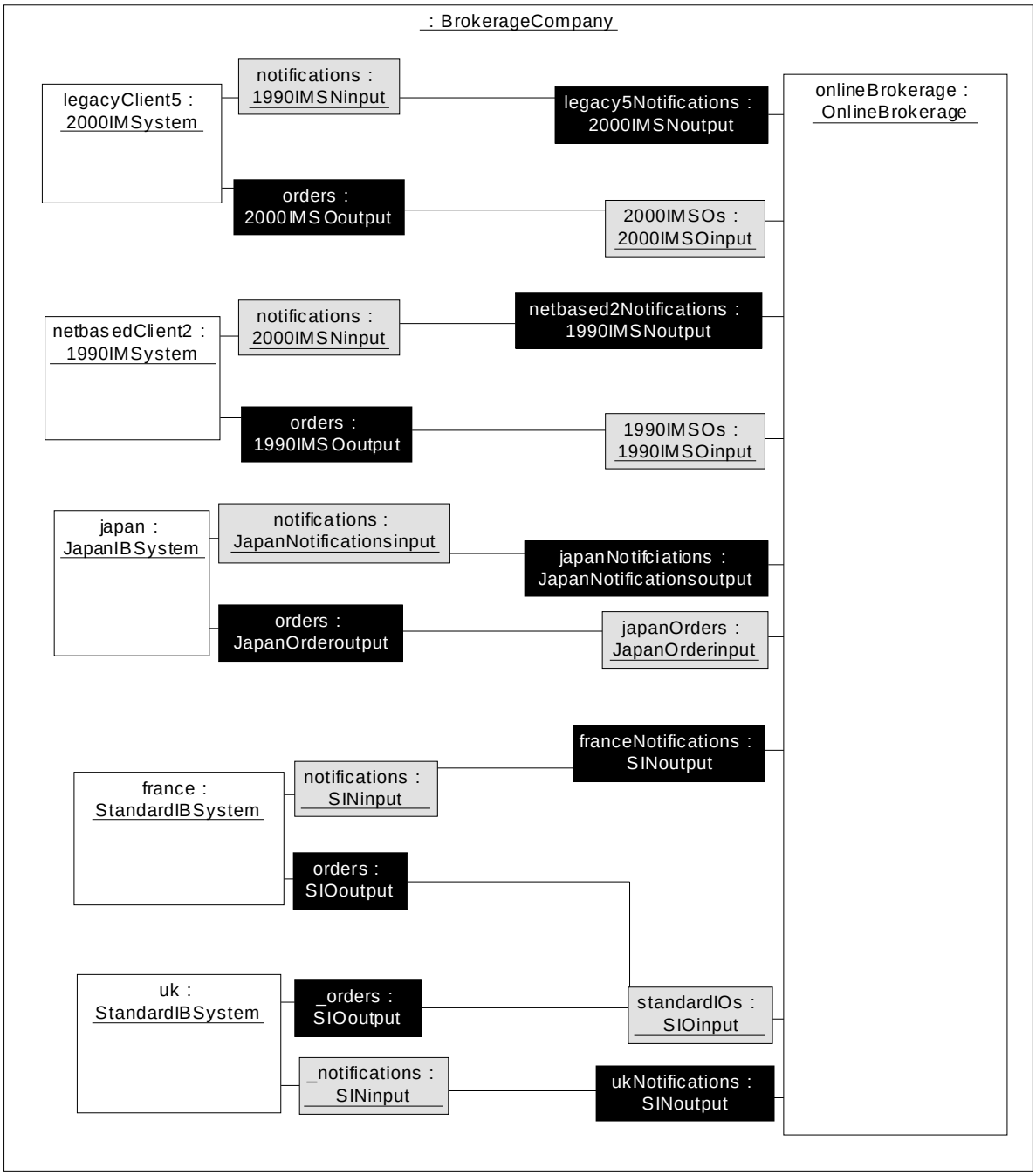


Figure 8-26 Collaboration diagram for example with components of same type.

This example happens to illustrate the top-level definition of an EAI architecture, in this case for a brokerage company.

8.3.12.4 Call and Request/Reply Adapters

A common configuration of components is the connection of call and request/reply adapters. This is illustrated by Figure 8-27.

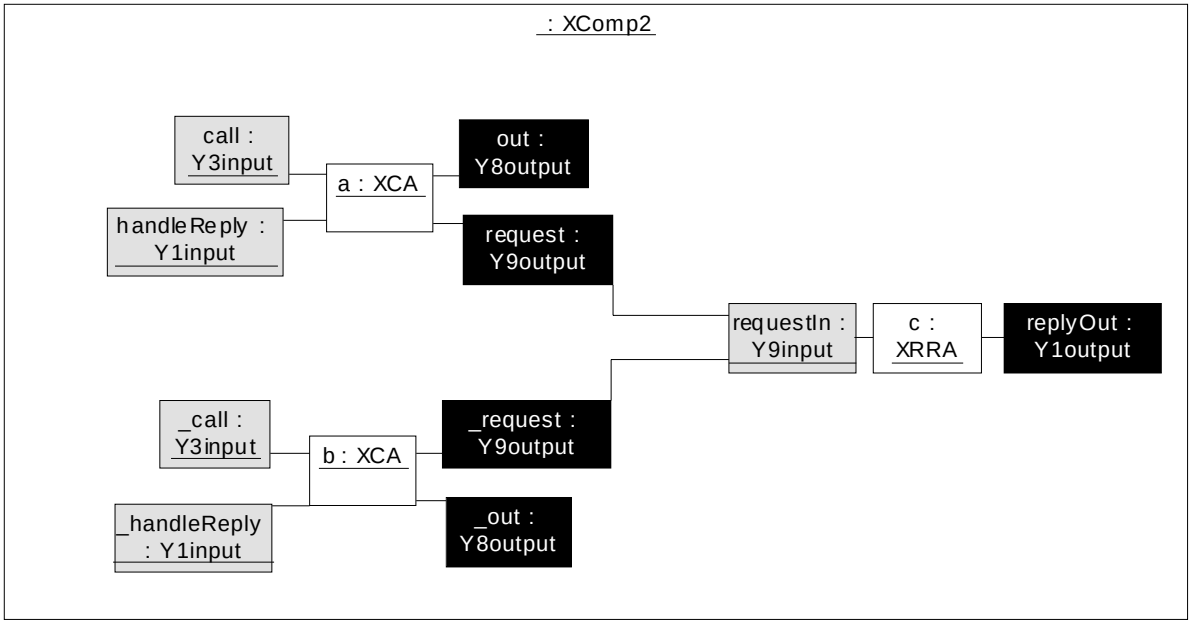


Figure 8-27 Configuration of call and request/reply adapters

Here, two call adapters (a and b) are connected to a single request/reply adapter (c). The call adapters get information from an underlying system through their call terminals. They construct requests that are then passed on to the requestIn terminal of the request/reply adapter. This processes the request, usually by making a call to some underlying system, and then constructs a reply, which it puts on its replyOut terminal. Before sending the reply, the original request is examined to identify the terminal to which the reply must be sent (which will be the handleReply terminal for a or b, depending on which one sent the request), and this is added to the target terminals list of replyOut, just for the duration of sending the reply.

8.3.12.5 Publish and Subscribe

Another common configuration of components is the connection of publication and subscription operators. This is illustrated by Figure 8-28.

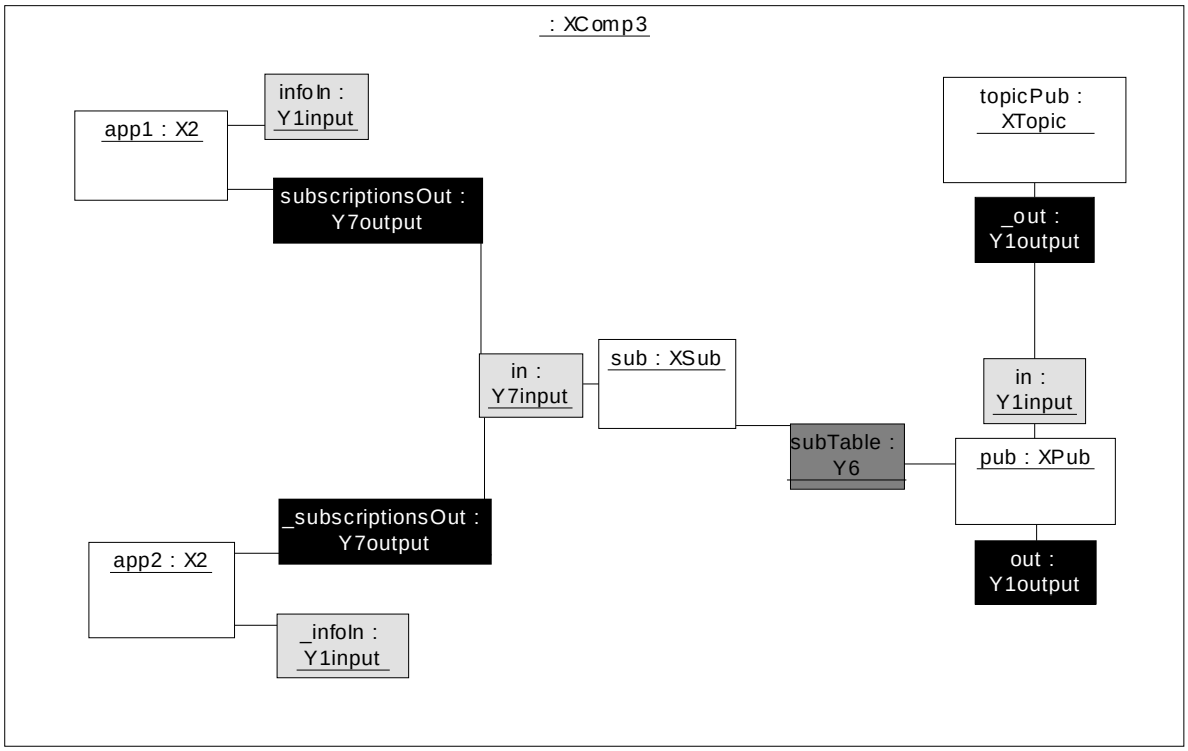


Figure 8-28 Configuration of publication and subscription operators

A publication operator pub is fed information to publish by a topic publisher topicPub. The feed is provided by the connection of the out terminal of topicPub to the in terminal of pub. Now pub has a subscription table (subTable) which it shares with the subscription operator sub. Two applications, app1 and app2, send subscription requests to sub. The subscription requests will identify their infoIn terminals as the terminals where published information, matching the criteria of the subscriptions, should be received.

A more sophisticated (and more common) version of this example would have multiple topic publishers feeding messages to the publication operator. Then multiple publishers would share the subscription table of the subscription operator.

8.3.12.6 Constraints

Only operators with stereotype <<Compound>> can have composition associations, and these must be with other operators (classes with an operator stereotype). The associations have a label but no indication of cardinality.

The type of content of the terminals must have a stereotype <<MessageContent>> or one of its substereotypes.

The class and collaboration diagrams used to notate a compound operator must be consistent. This means:

- Names of terminal objects must match the labels on terminal associations on the class diagram. The types of the object must correspond to the terminal classes defined in the class diagram.
- Names of component operator objects must match the labels on the composition associations on the class diagram. The types of the objects must correspond to the operator classes at the target of those associations as defined on the class diagram.

On the collaboration diagram, only output terminals may be connected to input terminals of other components. Input (output) terminals of the compound operator may only be connected to input (output) terminals of components.

The content type handled by terminals must be the same for any two terminals connected together on the collaboration diagram.

8.4 Adapters

8.4.1 Source Adapters

Figure 8-1 shows the general format of the notation used to define a source adapter, which is represented by a class with stereotype <<SourceAdapter>>. A source adapter is an operator that obtains information from a system (e.g., vendor-supplied package or legacy application system), where that information might not be in a message content format, translates it into message content of a given output type and then sends out a message with that content.

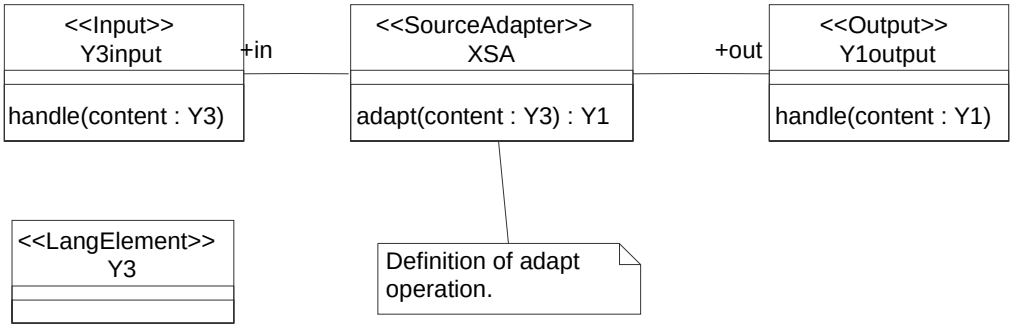


Figure 8-1 Class diagram for prototypical source adapter

When using a source adapter as a component of a compound operator (see Section 8.3.12, “Compound Operators,” on page 8-14), it is usually the case that its input terminal will not be connected to any other terminals. How information gets placed on that terminal is left unstated, since the internals of an application are out of scope for EAI modeling.

Constraints

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the adapt operation.

The type of content of the output terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the input terminal must have a stereotype <<LangElement>> or one its substereotypes.

8.4.2 Target Adapters

Figure 8-2 shows the general format of the notation used to define a target adapter, which is represented by a class with stereotype <<TargetAdapter>>. A target adapter is an operator that accepts messages and translates them into information for a system (e.g., vendor-supplied package or legacy application system), where that information might not be in a message content format.

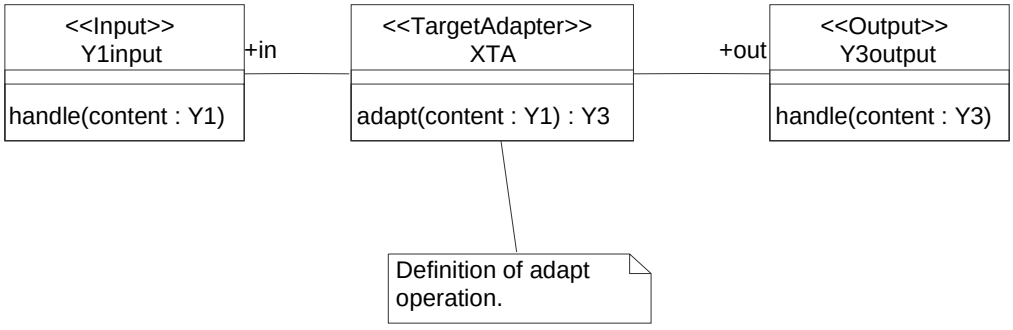


Figure 8-2 Class diagram for prototypical target adapter

When using a target adapter as a component of a compound operator (see Section 8.3.12, “Compound Operators,” on page 8-14), it is usually the case that its output terminal will not be connected to any other terminals. What happens to information after it leaves that terminal is left unstated, since the internals of an application are out of scope for EAI modeling.

Constraints

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the adapt operation.

The type of content of the input terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the output terminal must have a stereotype <<LangElement>> or one its substereotypes.

8.4.3 Call Adapters

Figure 8-3 shows the general format of the notation used to define a call adapter.

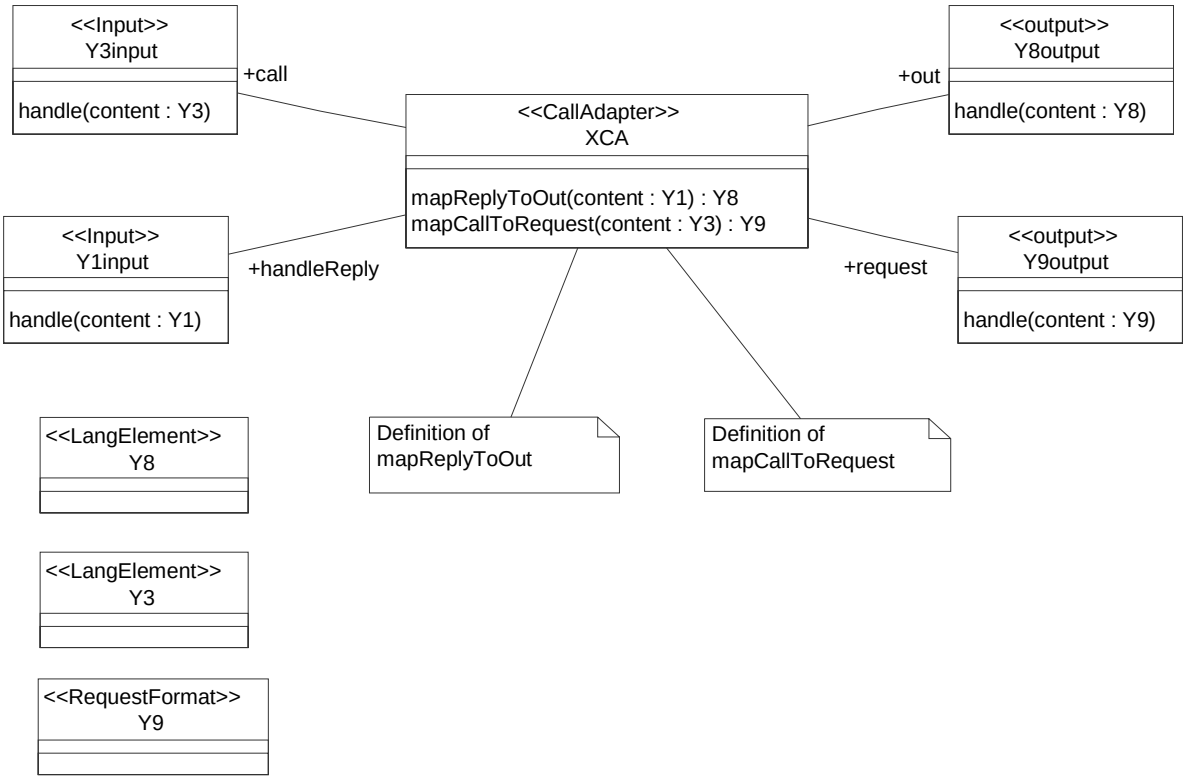


Figure 8-3 Class diagram for prototypical call adapter

A call adapter is invoked synchronously by an application that wishes to make use of a service (made available via a server) that can respond to a request message and send a response message back to the service requester. It accepts a call (which is not in a standard message format) on its call terminal and maps that call to a request message, which it sends to the request terminal. On receipt of a reply from the handleReply terminal, it maps that reply to a format understood by the application and places the result of the mapping on the out terminal.

A call adapter is used in conjunction with a request/reply adapter. See Section 8.3.12.4, “Call and Request/Reply Adapters,” on page 8-21 for details.

Constraints

The input terminals must be labeled call and handleReply, and the output terminals out and request.

The type of content of call and request must match the type of the parameter and result, respectively, of the mapCallToRequest operation.

The type of content of handleReply and out must match the type of the parameter and result, respectively, of the mapReplyToOut operation.

The type of content of the handleReply terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the call and out terminals must have a stereotype <<LangElement>> or one its substereotypes. The type of content of the request terminal must have a stereotype <<RequestFormat>>.

8.4.4 Request/Reply Adapters

Figure 8-4 shows the general format of the notation used to define a request/reply adapter.

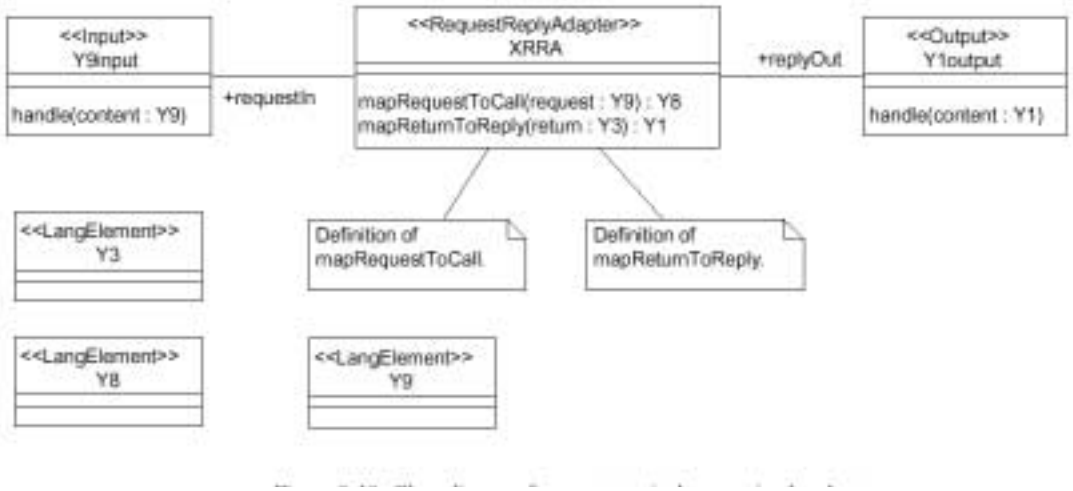


Figure 8-4 Class diagram from prototypical request/reply adapter

A request/reply adapter receives a request (from a call adapter) which contains both a terminal identifier and some other content. The mapRequestToCall operation extracts the information content of the request and converts it to a format suitable for passing to some underlying system. The mapReturnToReply operation takes the information returned from the system and constructs a message which is placed on the output terminal, but only after the terminal identifier in the original request has been added to the target list of its replyOut terminal. When the message has been sent, the terminal identified in the request message is removed from the target set of replyOut.

Note that any terminal permanently connected to the replyOut terminal will have replies of all requests broadcast to it.

A request/reply adapter is used in conjunction with a call adapter. See Section 8.3.12.4, “Call and Request/Reply Adapters,” on page 8-21 for details.

Constraints

The input terminal must be labeled requestIn, and the output terminal replyOut.

The type of content of requestIn and replyOut must match the type of the parameter of mapRequestToCall and the result of mapReturnToReply, respectively.

The type of content of the replyOut terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of content of the requestIn terminal must have a stereotype <<RequestFormat>>. The type of the result of mapRequestToCall and the parameter of mapReturnToReply must have a stereotype of <<LangElement>> or one of its substereotypes.

8.5 Sources and Sinks

8.5.1 Sources and Queued Sources

Figure 8-5 shows the general format of the notation used to define a source, which is represented by a class with stereotype <<Source>>. A source is an operator that delivers message content to an output terminal. How that message content is constructed, or where it comes from, is not stated.

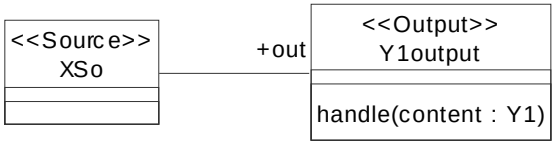


Figure 8-5 Class diagram for prototypical source

A queued source is a source that has a <<Queue>> resource. It is identified by the stereotype <<QSource>>, as illustrated by Figure 8-6 .

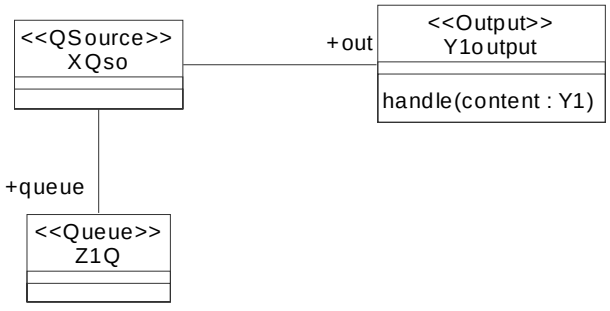


Figure 8-6 Class diagram for prototypical queued source

Constraints

The type of content of the output terminal must have a stereotype *<<MessageContent>>* or one of its substereotypes.

For queued sources, there must be a directed association to a queue resource (i.e., a class with stereotype *<<Queue>>*). This should be labeled *queue*.

8.5.2 Sinks and Queued Sinks

Figure 8-7 shows the general format of the notation used to define a sink, which is represented by a class with stereotype *<<Sink>>*. A *sink* is an operator that receives message content from an input terminal. What happens to that content thereafter is left unsaid.

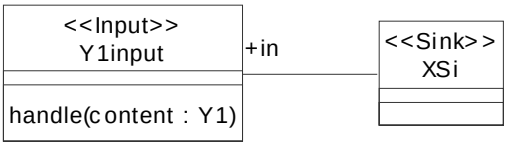
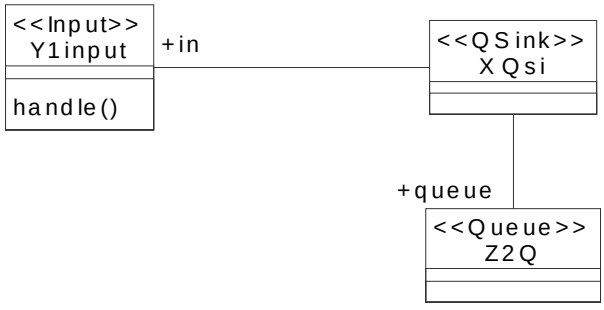


Figure 8-7 Class diagram for prototypical sink

A queued sink is analogous to a queued source, and is identified by the stereotype <<QSink>>.



Constraints

The content format of the input terminal must have a stereotype <<MessageContent>> or one of its substereotypes.

For queued sinks, there must be a directed association to a queue resource (i.e., a class with stereotype <<Queue>>). This should be labelled queue.

8.6 Resources

Resources are things that operators use to do their job, but which are not themselves operators. The specific resources declared in this profile are databases, queues and subscription tables.

Resources are defined as classes with stereotype <<Resource>> or one of its substereotypes: <<Database>>, <<Queue>> and <<SubscriptionTable>>.

The use of a resource by an operator is indicated, in the class diagram defining that operator, by a directed association from the operator to the resource. See Section 8.3.2, “Transformers and Database Transformers,” on page 8-4 and Section 8.3.9, “Subscription Operators,” on page 8-12 for examples.

When operators with resources are used as part of a compound, they may share a resource. This is shown by adding an object of the resource class and connecting the sharing operators to it with a link. See Section 8.3.12.5, “Publish and Subscribe,” on page 8-21 for an example.

8.7 Message Formats

8.7.1 MessageContent Core

The data contained in a message is its MessageContent. Messages are defined using ordinary UML class modeling mechanisms. However, message content classes are restricted to represent transmittable data structures.

The model for messages is that they may contain one or more parts, each of which may have its own header part. The header contains information used by the messaging infrastructure to control how it deals with the message. Each message part may also have a body section, which contains the application data. Message parts may be nested.

Both the header and the body may contain nested structures of primitive message elements.

We formalize these restrictions using the UML stereotypes given in Table 8-1. A class of the <<MessageContent>> stereotype represents a serialized message. To reflect the ordering of the parts of a message, there are additional constraints:

1. All associations are ordered with respect to each other.
2. Associations of multiplicity greater than one are ordered.

For example, a message header usually occurs in a message part before the message content.

Table 8-1 Stereotype specification for message content description

Stereotype	Parent	Tags	Constraints	Description
MessageContent	N/A	domainformat	May only have containment associations with classes of stereotype <<MessagePart>> or <<ComposedMessagePart>>	Top level for describing messages (such as a MIME envelope)
MessagePart	N/A	NA	May only be composed by a class of stereotype <<MessageContent>>May contain a 'header' association with a class of stereotype <<MessageElement>>May contain a 'body' association with a class of stereotype <<MessageElement>>	Used to describe 'large scale' message structuring (such as MIME parts)
ComposedMessagePart	MessagePart	NA	May have associations with classes of stereotype <<MessagePart>> and of stereotype <<ComposedMessagePart>>	Used to describe nested message parts
LangElement	NA	NA		Models message headers, message bodies and their content

Figure 8-29 shows an example of a content class with two data items, an integer and a string. These simple message parts have been rendered as attributes of the owning SimpleContent class. This is recommended in order to allow compact representation of simple message types.

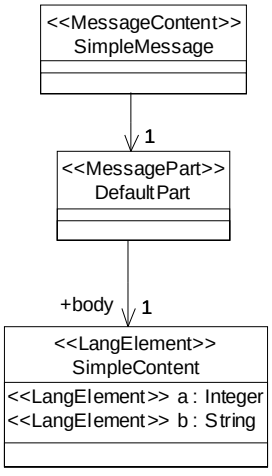


Figure 8-29 A simple message content class

More complicated message-content structures can be created using composition, as is shown in Figure 8-30. This models a message which has a single part. The message has as its header a string, while the message body is a table of addresses. This table has a single integer, records, that is a count of the records in the message.

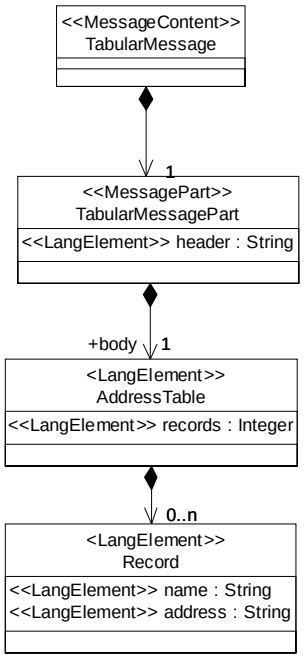


Figure 8-30 A model of a message containing a table

8.7.2 Basic MOM Message Structure

The stereotypes given in the preceding section provide the framework to allow messages to be specified, but they do not cover commonly occurring concepts supported by message oriented middleware (MOM) products.

In this section we add the basic concept of an exception message, a message sent by the messaging infrastructure when a fault occurs in the processing of a message. We also define a MOMHeader, which can specify an exception target (the location to which a message should be sent in the event of an exception) and a reply target, and it can identify the kind of message being sent.

Table 8-2 Stereotype specification for MOM structure

Stereotype	Parent	Tags	Constraints	Description
MOMHeader	MessageElement	NA	May have an association 'replyTo' with a <<MessageElement>> class that specifies a reply target and another 'exceptionTarget' with a <<MessageElement>> class that specifies an exception target	Stereotype to capture common MOM header information
ExceptionNotice	MessageContent	NA	May have a message part containing the header and body of the message that caused the fault	Message sent by the MOM infrastructure if a fault occurs while processing a message

8.7.2.1 ExceptionNotice

Figure 8-31 illustrates the usage of the ExceptionNotice stereotype. In this example, we have defined a class MOMException, which models the message content of an exception message created by a MOM system after a fault has occurred. MOMException contains two associations to classes that conform to the MessagePart stereotype:

- originalMessage is an association to a class that models the content of the message that caused the exception. In this case, the original message had just one message part. If the original message had contained several parts, it would be possible to model originalMessage as a class that conforms to the ComposedMessagePart stereotype.
- exceptionInformation is an association to a message part that contains only exception header information. The exception header holds information that identifies the exception type and a string that describes the exception.

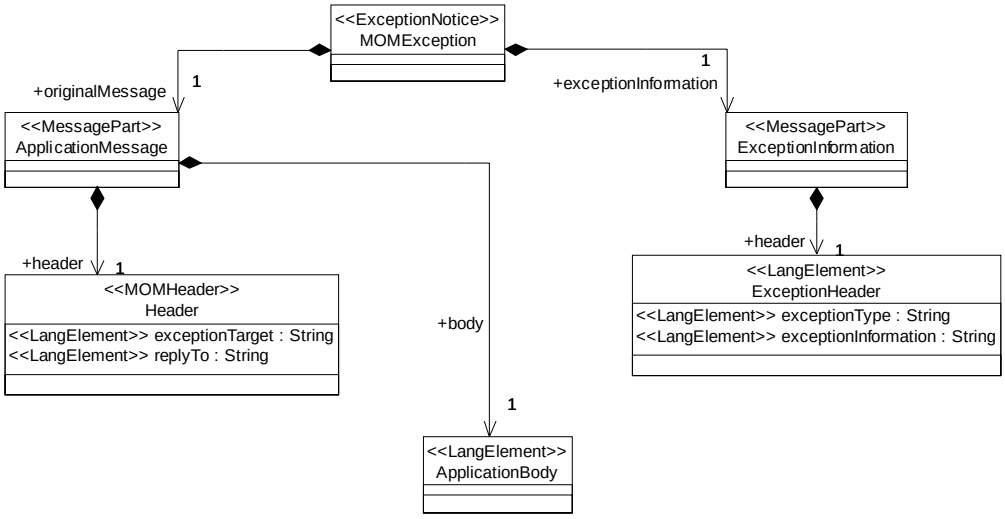


Figure 8-31 Example of the use of the ExceptionNotice and MOMHeader stereotypes

8.7.2.2 MOMHeader

The MOMHeader stereotype demands that a message header must identify the following elements, but does not dictate how they are represented in the message:

- replyTo: a means of identifying a location to send a reply message to
- exceptionTarget: a means of identifying a location to send an exception notice in the event of a fault occurring in the processing of a message

Figure 8-32 demonstrates an example of the use of the MOMHeader stereotype. In this case, the domain and format are both identified using strings, and the exceptionTarget and replyTo header content are specified using the MOMEndpointSpec class. In a particular MOM implementation, this information should allow an EAI terminal to be identified.

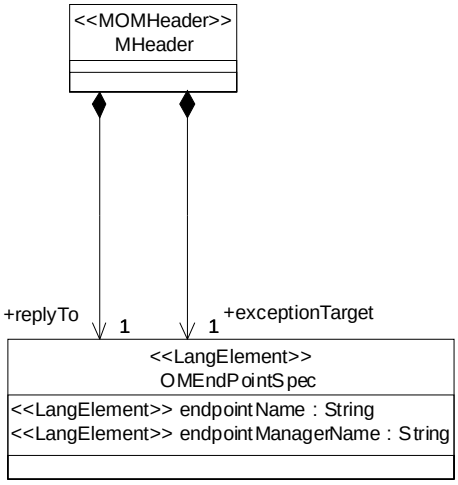


Figure 8-32 Example of the use of the MOMHeader stereotype

8.8 Mapping with Metamodel

The mapping with the metamodel is summarized by a series of tables, which are organized below into sections corresponding to the four main parts of the profile: terminals, operators, resources and message formats.

These tables are based on the approach specified in UML 1.4. for defining stereotypes for use in a profile. We have extended them to show the mapping to the EAI metamodel. Thus the tables also serve to summarize the stereotypes used in the profile.

In addition to the tables, we have detailed important mapping constraints which dictate how information associated with an instance of an EAI metaclass is related to information associated with an instance of the stereotyped UML base class. These are listed below the relevant tables.

The mapping constraints should be distinguished from constraints that apply to the use of the profile itself (e.g., the use of a particular stereotype). Those are defined in the section describing that aspect of the profile.

8.8.1 Terminals

Table 8-3 Mapping of terminals

EAI Metaclass	Base class	Stereotype	Parent	Description & constraints
EAITerminal	Core::Association			See Section 8.2
	Core::Class	Input or Output		See Section 8.2
EAIQueuedInputTerminal	Core:: Association			See Section 8.2
	Core::Class	QInput	Input	See Section 8.2
EAIQueuedOutputTerminal	Core:: Association			See Section 8.2
	Core::Class	QOutput	Output	See Section 8.2

Mapping Constraints

EAITerminal

1. This mapping is valid only for terminals which belong to operators that define types.
2. The association is sourced on the class corresponding to the operator to which the terminal belongs; it is targeted on the class identified with the terminal.
3. The handle operation of the class must have a parameter of a type corresponding to the type of the parameter associated with the terminal.
4. Different terminals may map to the same class (but not the same association).
5. The name of the terminal is the name of the target end of the association.
6. The stereotype of the class corresponds to the value of the terminalKind attribute of the terminal.

EAIQueuedInputTerminal and EAIQueuedOutputTerminal

There are no additional constraints.

8.8.2 Operators

Operators and terminals in the metamodel are used in two roles. Firstly they are used to define types and parameters; secondly they are used to define the connectivity of a compound operator in its role in defining a type. The mapping of operators has been split into two parts, reflecting the two different roles. The first part deals with all operators, except compound operators. The second part deals with compound operators, which, as suggested above, requires a second mapping of operators and terminals to be defined.

Table 8-4 Mapping of operators (except compound)

EAI Metaclass	Base class	Stereotype	Parent	Description & constraints
EAIPrimitiveOperator	Core::Class	PrimitiveOperator		See Section 8.3.1
EAITransformer	Core::Class	Transformer	PrimitiveOperator	See Section 8.3.2
EAIDBTransformer	Core::Class	DBTransformer	Transformer	See Section 8.3.2
EAIFilter	Core::Class	Filter	PrimitiveOperator	See Section 8.3.3
EAIStream	Core::Class	Stream	PrimitiveOperator	See Section 8.3.4
EAIPostDater	Core::Class	PostDater	Stream	See Section 8.3.5
EAIAggregator	Core::Class	Aggregator	PrimitiveOperator	See Section 8.3.12
EAISubscriptionOperator	Core::Class	SubscriptionOperator	PrimitiveOperator	See Section 8.3.15
EAIPublicationOperator	Core::Class	PublicationOperator	PrimitiveOperator	See Section 8.3.16
EAITopicPublisher	Core::Class	TopicPublisher	PrimitiveOperator	See Section 8.3.17

Mapping Constraints

EAIPrimitiveOperator

7. This mapping is only valid for primitive operators defining a type (not ones used to show connectivity of components).
8. The name of operator (and hence the type which the operator defines) is the name of the class.
9. There must be an association on the class diagram corresponding to each terminal of the primitive operator.

EAITransformer

10. The transformation mapping of the operator maps to the operation transform, in the class corresponding to the operator.

EAIDBTransformer

11. The database resource maps to the database association sourced on the class corresponding to the operator.

EAIFilter

12. The filterCondition of the operator maps to the allow operation in the corresponding class.

EAIStream

13. The emissionCondition of the operator maps to the emit operation in the corresponding class.

EAIPostDater

14. The timerMapping of the operator corresponds to the setTimingCondition operation in the corresponding class.

EAIAggregator

15. The aggregateComplete condition of each EAIMessageAggregation of the operator corresponds to the aggregateComplete operation in the corresponding <<MessageAggregation>> class.

16. The addToAggregate condition of each EAIMessageAggregation of the operator corresponds to the addToAggregate operation in the corresponding <<MessageAggregation>> class.

17. The aggregationMapping of each EAIMessageAggregation of the operator corresponds to the aggregate operation in the corresponding <<MessageAggregation>> class.

EAISubscriptionOperator and EAIPublicationOperator

18. The subscriptionTable resource maps to the subscriptionTable association sourced on the class corresponding to the operator.

TopicPublisher

There are no further constraints.

A compound operator utilizes a graph of operators, terminals and resources to define the connectivity of its components. This is exposed by the mapping defined in Table 6.

Table 8-5 Mapping of compound operator

EAI Metaclass	Base class	Stereotype	Parent	Description & constraints
EAICompoundOperator	Core::Class	CompoundOperator		See Section 8.3.18
	CommonBehavior::Object			See Section 8.3.18
EAITimer	Core::Class	Timer	CompoundOperator	See Section 8.3.13
EAIRouter	Core::Class	Router	CompoundOperator	See Section 8.3.14
EAIPrimitiveOperator (and subclasses)	Core::Association, CommonBehavior::Object			See Section 8.3.18
EAICompoundOperator (and subclasses)	Core::Association, CommonBehavior::Object			See Section 8.3.18
EAIResource (and subclasses)	CommonBehavior::Object			See Section 8.3.18
EAITerminal (and subclasses)	CommonBehavior::Object			See Section 8.3.18
EAILink	CommonBehavior::Link			See Section 8.3.18

Mapping Constraints

EAICompoundOperator

19. This mapping is only valid for compound operators defining a type (not ones used to show connectivity of components).

20. The name of operator (and hence the type which the operator defines) is the name of the class.

21. There must be an association on the class diagram corresponding to each terminal of the compound operator.

22. On the class-diagram part of the definition of the compound operator, there must be an association for each component operator.

23. The object is unnamed on the collaboration diagram defining the connectivity of the compound's components, and it contains all objects corresponding to the component operators and their terminals.

24. On the collaboration diagram, the objects corresponding to the terminals of the operator appear outside the object corresponding to the operator.

EAITimer and EAIRouter

25. Exceptionally, the components of these operators are not exposed in the profile. Therefore they do not have collaboration diagrams associated with them, and they do not map to objects.

EAIPrimitiveOperator (and subclasses), EAICompoundOperator (and subclasses).

26. This mapping is only valid for operators which are used in the role of defining the components of a compound operator. That is, they do not define a type, and they are owned by a compound operator (one of its nodes).

27. The association must be a composite association. The name of the part end corresponds to the name of the operator. The association is sourced on the class corresponding to the compound operator of which the operator in question is a part, and targeted on the class corresponding to the operator which defines the type of the operator in question.

28. The name of the object corresponds to the name of the operator. The type of the object is the class that corresponds to the operator which defines the type of the operator in question.

29. There must be an object corresponding to each terminal of the operator, and this must be linked to the object corresponding to the operator.

30. The object corresponding to the operator in question may be linked to an object corresponding to a resource, if the operator that defines the type of the operator in question is associated with a resource. The type of the resource object is the class corresponding to the resource.

EAIResource (and subclasses)

31. This mapping is only valid if the resource is associated with an operator used in the role of defining a component of a compound.

EAITerminal

32. The name of the object is the name of the terminal. The type of the object is the class corresponding to the terminal that defines the parameter associated with the terminal.

EAILink

33. The (UML) link must connect the objects associated with terminals that the (EAI) link connects.

34. The (UML) link has no message if the value of the synchronization attribute of the (EAI) link is unspecified. It has a synchronous (asynchronous) message if the value of that attribute is synchronous (asynchronous). The direction of the message is from the object corresponding to the source of the (EAI) link, to the object corresponding to the target of the (EAI) link.

8.8.3 Adapters

Table 8-6 Mapping of adapters

EAI Metaclass	Base class	Stereotype	Description & constraints
EAISourceAdapter	Core::Class	SourceAdapter	See Section 8.3.6
EAITargetAdapter	Core::Class	TargetAdapter	See Section 8.3.7
EAICallAdapter	Core::Class	CallAdapter	See Section 8.3.8
EAIRequestReplyAdapter	Core::Class	RequestReplyAdapter	See Section 8.3.9

Mapping Constraints

EAISourceAdapter and EAITargetAdapter

35. The internalToMessage (resp. messageToInternal) mapping for the operator corresponds to the adapt operation in the corresponding class.

EAICallAdapter

36. The callToRequestMapping of the operator corresponds to the mapCallToRequest operation in the corresponding class.

37. The replyToOutMapping of the operator corresponds to the mapReplyToOut operation in the corresponding class.

EAIRequestReplyAdapter

38. The requestToCallMapping of the operator corresponds to the mapRequestToCall operation in the corresponding class.

39. The returnToReplyMapping of the operator corresponds to the mapReturnToReply operation in the corresponding class.

8.8.4 Sources and Sinks

Table 8-7 Mapping of Sources and Sinks

EAI Metaclass	Base class	Stereotype	Description & constraints
EAISource	Core::Class	Source	See Section 8.3.10
EAIQueuedSource	Core::Class	QSource	See Section 8.3.10
EAISink	Core::Class	Sink	See Section 8.3.11
EAIQueuedSink	Core::Class	QSink	See Section 8.3.11

Mapping Constraints

EAISource, EAIQueuedSource, EAISink, EAIQueuedSink

There are no further constraints.

8.8.5 Resources

Table 8-8 Mapping of resources

EAI Metaclass	Base class	Stereotype	Parent	Description & constraints
EAIResource	Core::Association			See Section 4.4
	Core::Class	Resource		See Section 4.4
EAIDatabase	Core:: Association		Resource	See Section 4.4
	Core::Class	Database	Resource	See Section 4.4
EAIQueue	Core:: Association		Resource	See Section 4.4
	Core::Class	Queue	Resource	See Section 4.4
EAISubscriptionTable	Core:: Association		Resource	See Section 4.4
	Core:: Class	SubscriptionTable	Resource	See Section 4.4

8.8.5.1 Mapping Constraints

40. The name of the resource maps to the name of the target end of the association.

41. The source of the association is the class corresponding to the operator associated with the resource.

42. The target of the association must be a class with a stereotype corresponding to the name of the (metamodel concrete) class of the resource.

8.8.6 Message Formats

Table 8-9 Mapping of formats

EAI Metaclass	Base Class	Stereotype	Parent	Description & constraints
EAIMessageContent	Core::Class	MessageContent		See Section 4.5.1
EAIMessagePart	Core::Class	MessagePart		See Section 4.5.1
EAIComposedMessagePart	Core::Class	ComposedMessagePart	MessagePart	See Section 4.5.1
TDLangElement	Core::Class	LangElement		See Section 4.5.1
EAIHeader	Core::Class	MOMHeader	MessageElement	See Section 4.5.2
EAIExceptionNotice	Core::Class	ExceptionNotice	MessageContent	See Section 4.5.2

Mapping Constraints

TDLangElement

43. Composed types will map to a TDLangElement with a TDLangComposedType. See Section 7.3.8.4, “TDLangElement,” on page 7-21.

Activity Modeling 9

Messages are produced as a result of business events occurring in enterprise applications. The sequence of these events and the resulting message flows across system boundaries is defined in the overall system integration process. This section describes a profile for modeling EAI processes using activity graphs. These models can subsequently be refined to realize the functionality specified using the stereotypes defined in Chapter 8.

9.1 Modeling Integration Processes

Chapter 8 describes a profile for defining the collaborations necessary for application integration. It may be characterized as a profile for designing integrations. Many application-integration developers also adopt a process-oriented approach where the initial artifact is a definition of the business process, end-to-end, which is to be integrated. Of course such a process definition will encompass many integration points, each of which will need to be implemented. The value of the process view is to establish the requirement in a form that is understandable and verifiable by the business users. In this sense it is a requirements or analysis view and exists at a higher level of abstraction than the collaboration-based definitions of the previous chapter.

Whilst for any particular implementation approach it should be possible to map the analysis model onto the design model, it is beyond the scope of this specification to do so. In a sense, it would be pre-empting the development process. We consider a general formal mapping - with enforcement of a level of detail capable of formal mapping - to be inappropriate, since different practitioners have different approaches.

9.2 An Integration Process Scenario

Integration processes contain control flow and message flow aspects. Message flow is fundamental in EAI processes, as message-based integration is at the core of the problem domain.

This section introduces the profile elements required to define such models by means of an example scenario. Variants of the scenario are discussed. Some illustrate the capability of the profile to support high levels of abstraction, such as might be preferred for communicating with business users; others illustrate how the profile can be used to define more detail.

9.2.1 The Exchange Process

The scenario we have chosen is a collaborative business-to-business example, where Buyers and Sellers negotiate a transaction via an online Exchange. The overall process is represented in the activity graph in Figure 9-1. Annotations have been added (as parameters on transitions) to represent additional information about required operations (such as transformations) and to identify implementation details (such as queues). This is an example of where different practitioners might choose to capture this information at this level or may choose to omit it. The intention is that the profile is capable of representing it if required.

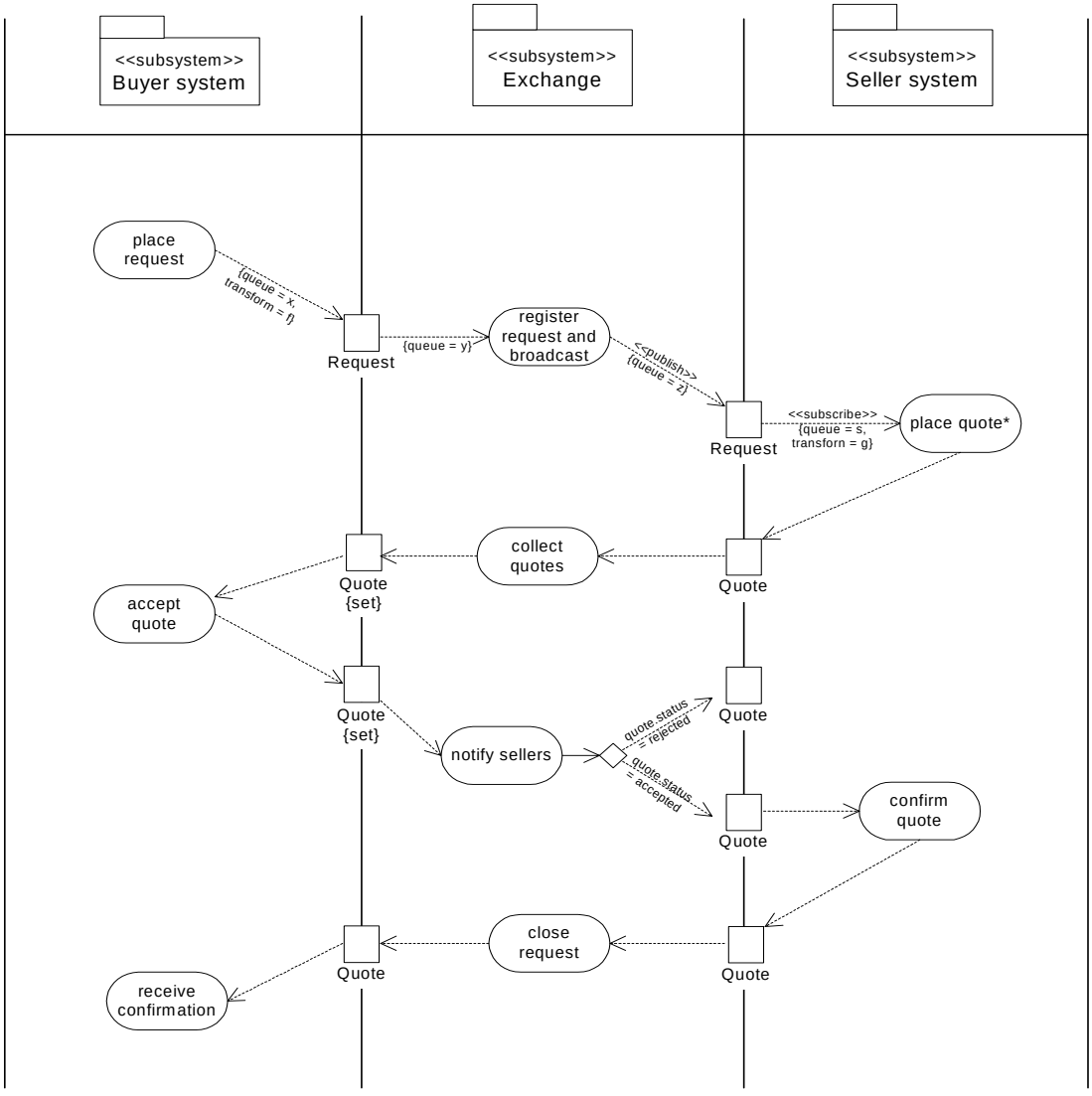


Figure 9-1 Basic way of modeling message based integration with Activities (Exchange example)

In this example, activities represent the legacy applications that consume and/or raise business events. The existence of connectors to detect and publish the events and to interpret these events for the legacy applications is implicit at this level of abstraction.

9.2.2 Modeling message flow explicitly

The message-flow aspects can be emphasized by using an explicit stereotype "messageFlow" for the transfer of messages between subsystems. This approach, illustrated in Figure 9-2, contrasts with the more abstract approach illustrated in Figure 9-1.

Note that the use of a transition between two ObjectFlowStates is not normal activity-graph usage, but is not prohibited by the UML semantics definitions.

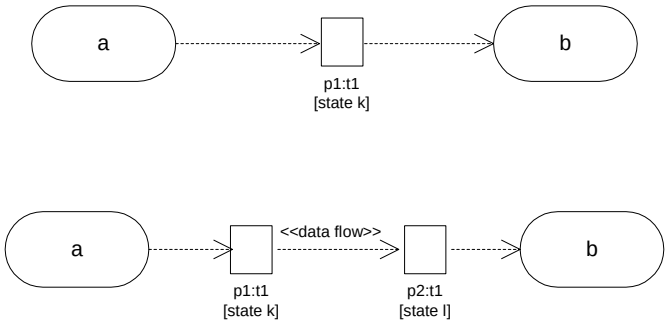


Figure 9-2 Application of the "messageFlow" stereotype to emphasize data-flow aspects

Figure 9-3 illustrates the impact of applying this technique to the exchange example.

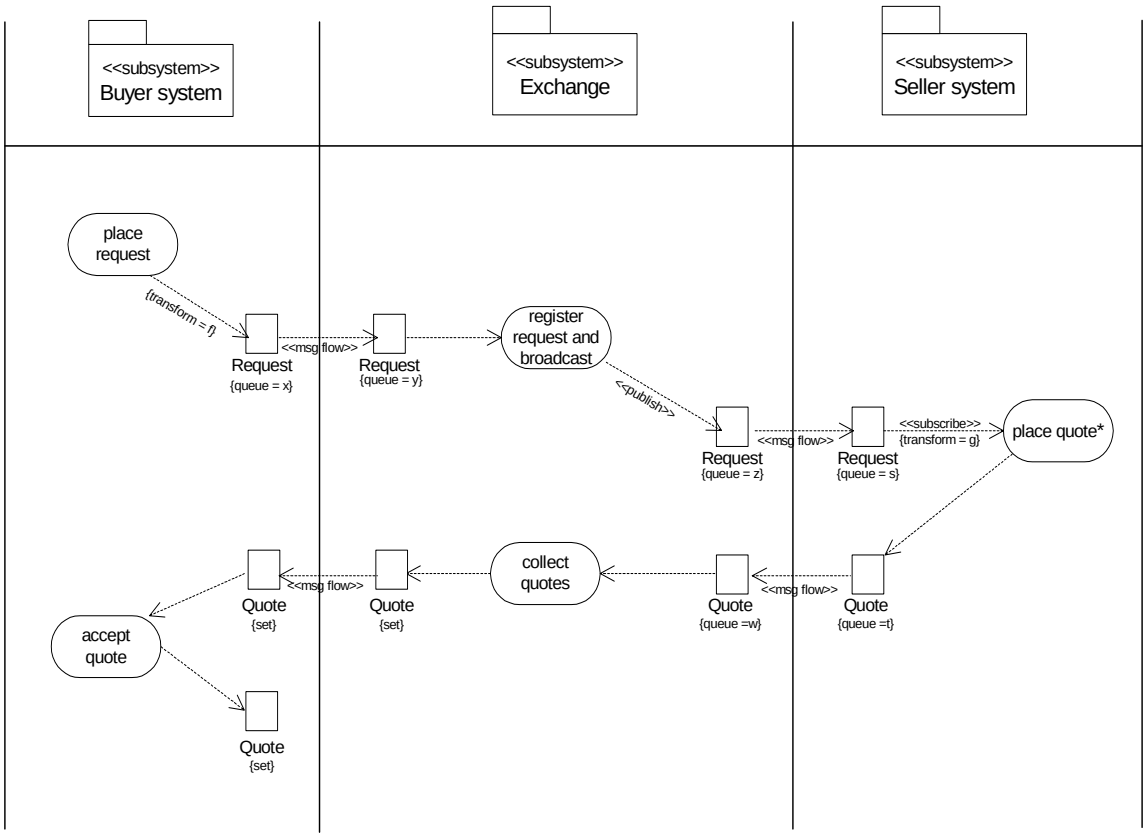


Figure 9-3 Application integration example with "messageflow" stereotype (partial)

In this activity graph, the partitions represent the enterprise systems that require integration. Inside each partition, action states (i.e., activities) represent the invocation of application APIs. A transition between an activity and an object-flow state represents the production or consumption of a message. Transitions with the stereotype "messageFlow" represent message transfers across system boundaries. Message flows may be "point-to-point." They may also be designated as "multicast" (according to a publish/subscribe protocol) by adding the stereotypes "publish" and "subscribe" to appropriate transitions.

9.2.3 Modeling control flow

In addition to message flow aspects, control flow aspects can be added to the process definition. In Figure 9-4, control flow transitions have been added within each of the component systems in a fragment of the Exchange example.

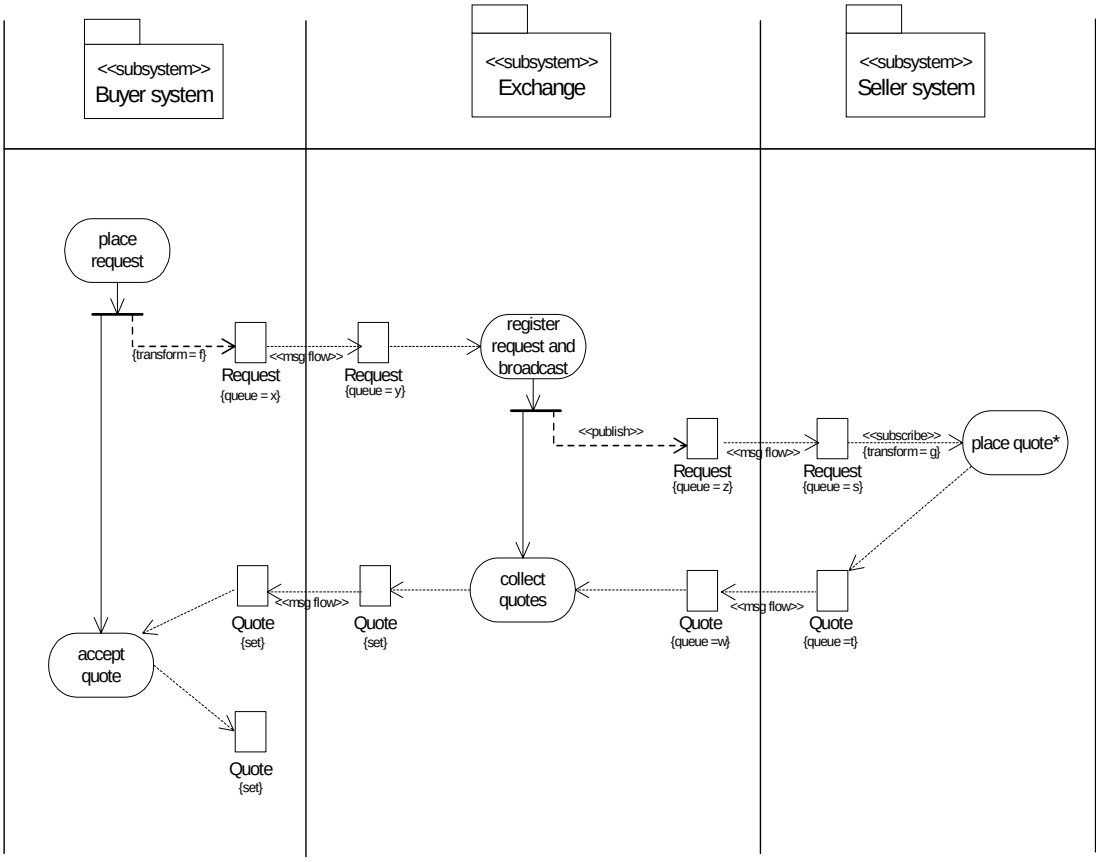


Figure 9-4 Optional control flow transitions between activities within a single system

9.2.4 Abstracting detail by decomposition

Activities can be decomposed to show the constituent set of subactivities. An example of the decomposition of the integration step "Place Quote" is shown in Figure 9-5. In this step, an incoming Request message results in an outgoing Quote message. In the decomposition, a "connector" activity is responsible for handling the incoming message. Once a message is accepted by the system, a "transformer" activity transforms the message content to a locally acceptable format. Finally, "adapter" activities take this known input and adapt it into the legacy data store format. A subsequent "adapter" is responsible for invoking the legacy system. After the legacy application has run, a similar set of steps produces the outgoing message.

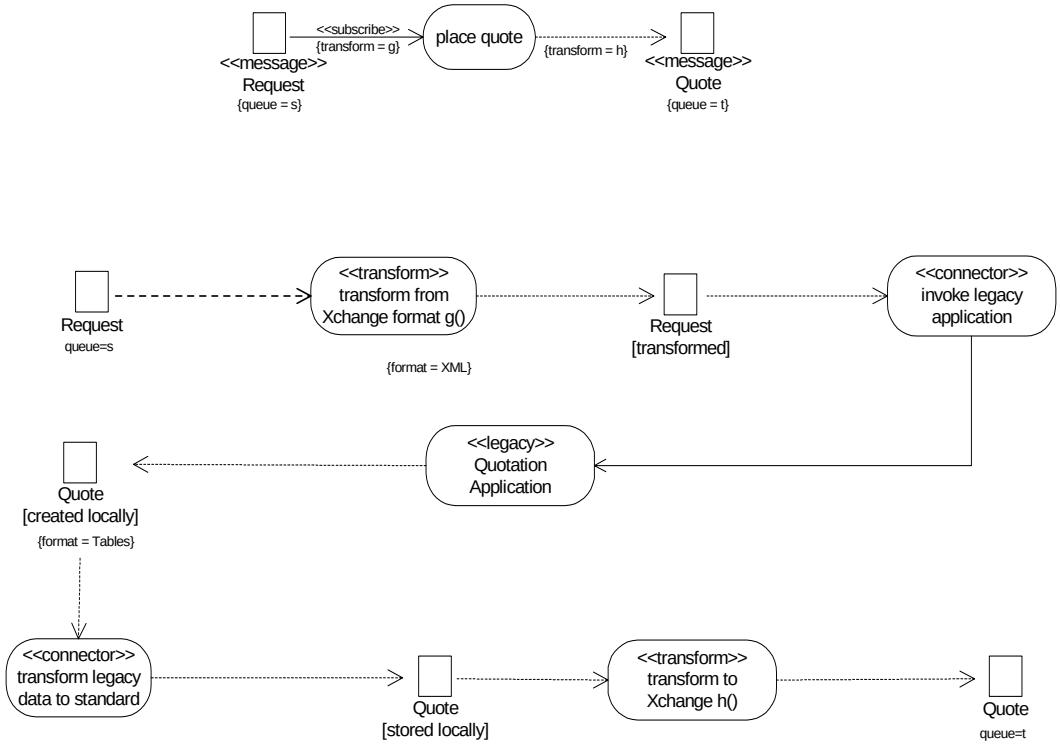


Figure 9-5 Decomposition of the integration step "Place Quote" in the context of the Exchange example

It should be noted that the above figure constitutes a prototypical example - many variants of these will exist using additional operator activities (e.g., involving "router" and "filter" operators) and with varying process structure.

9.2.5 Further fragmentary examples

Other activity graph constructions that can be used in modeling system-integration processes are described in the following subsections.

9.2.5.1 Multiple synchronized inputs and outputs

Multiple synchronized inputs and/or outputs can be modeled with join and fork pseudo-states (see Figure 9-6).

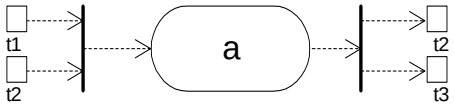


Figure 9-6 Modeling multiple inputs and outputs with join and fork pseudo-states.

9.2.5.2 Internal dataflows within a subsystem

An internal dataflow between two activities within a single system can be modeled with ObjectFlowStates (see Figure 9-7).

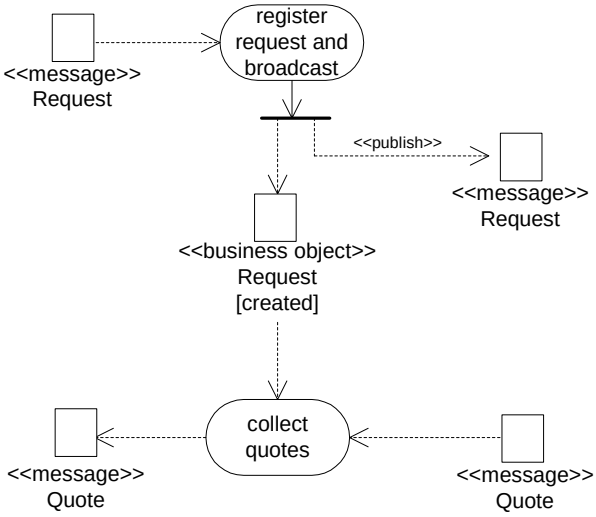


Figure 9-7 Modeling internal data flow with object flow states

9.2.5.3 Modeling decisions explicitly

Decisions can be modeled with guards, either implicitly with multiple outgoing or explicitly by using a decision PseudoState - the latter approach is relevant for modeling content based routing where the middleware is responsible for rule execution (as opposed to embedded rules executed by applications) (see Figure 9-8).

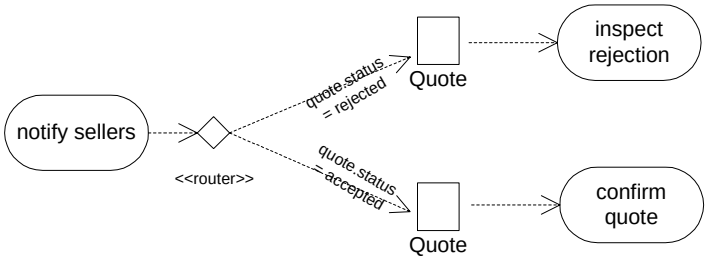


Figure 9-8 Example of a decision node to model rule-based routing

9.2.5.4 Synchronization

Synchronization is made explicit with Fork and Join PseudoStates - for instance, this can be used to model multiple parallel invocation of legacy systems in the case that there is more than one system (or more than one function) needing to be invoked (see Figure 9-9).

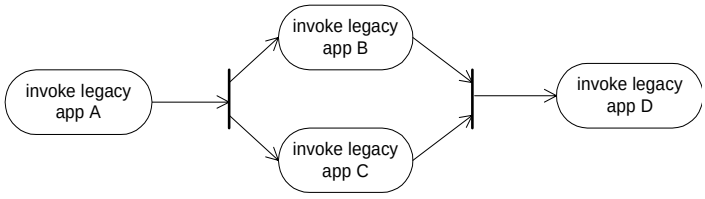


Figure 9-9 Synchronization with forks and joins

9.2.5.5 Multiple concurrent invocations of activities

Dynamic concurrent invocation of activities (where the number of actual activities invoked is determined at run time depending on the input) is denoted by a "*" symbol in the activity (see Figure 9-10).

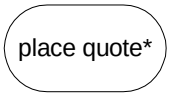


Figure 9-10 Dynamic concurrent invocation of an activity

9.2.5.6 Modeling business events explicitly

Events (as based on the definition of a signal in UML) can be added to transitions or they can be modeled explicitly as object-flow states. In the latter case, the underlying classifier is a signal, with attributes representing the event parameters. This can be

useful for modeling "adapter" implementations that respond to events (e.g., a database trigger) and for systems that natively expose a required integration event on their interface (see Figure 9-11).

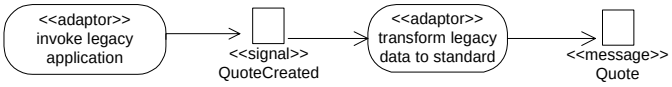


Figure 9-11 Explicit modeling of an event for an adapter implementation

Integration processes will usually not be defined beyond this level of detail in activity graphs. The design of the interactions between the classes involved is best described using collaboration modeling, as discussed in *Collaboration Modeling* chapter 8.

9.3 Profile Element Summary

The following is a summary of the activity-graph stereotypes and tagged values for modeling processes in the context of EAI. These stereotypes are in addition to the ones defined in Chapter 4. Tagged values on the activity stereotypes enable the linking of activities to their realization in terms of the Operator and Message Classifiers that implement the messaging functionality at the Collaboration level.

It should noted that some stereotypes apply to more than one UML metaclass. In some cases, the metaclass name is given in brackets to indicate that this is a secondary modeling option. For instance, a "transform" stereotype is primarily attached to an activity, indicating that the activity is realized by a transform operator. A "transform" stereotype can also be attached to a transition as a secondary option which can be useful for models that are (to be) decomposed.

This profile definition assumes the UML 1.3 extension mechanism, which is string-based. In UML 1.4, references to metaclasses can be used as an alternative to name based strings. Furthermore, in UML 1.4, multiple stereotypes can be applied to an element.

9.3.1 Stereotypes

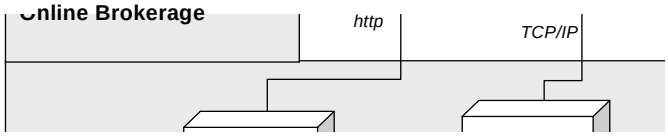
Table 9-1 defines the basic stereotypes. Tagged values for these stereotypes are defined in Table 9-2.

ActivityGraph	.. system integration process in the c
"ObjectFlowState	A data element that is interchanged b ObjectFlowState "inherits" the stereo with stereotype "message" that it poi

framework defined in Chapter 4. These references define messaging functionality specified in the integration proce diagrams).

Note: in the case of a subtype of "o
"connector", "publish" or "adapter"
transformationImplementation Clas
connectorImplementation Classifie

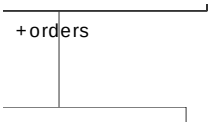
	.he context.
"FCMLink	The application of this stereotype is sp underlying queue implementation, but This is an example of where the analys

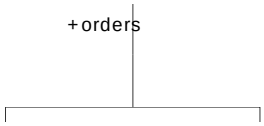

Online Brokerage
http
TCP/IP

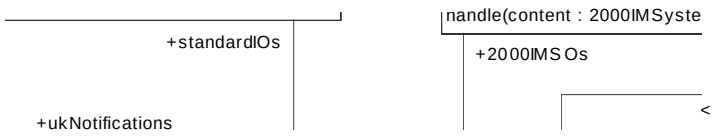

..otifications :
1990IMSNinput
legacy5Notifications :
2000IMSNoutput
onlin
Onl

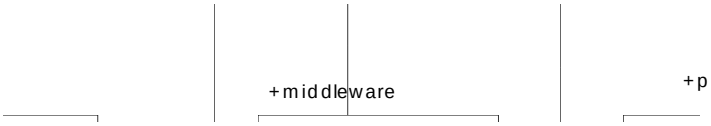
There are good business reasons (such as confidentiality) only notifications for France go to France and not to also

It has been left unspecified as to whether the connections


+orders


+orders


+standardIOs
+ukNotifications
nandle(content : 2000IMSyste
+2000IMSOs
<


+middleware
+p

system they are passed to the Pub/Sub server, which route to the IB or IM system, depending on which one generate systems generate notifications from the processed orders, external systems as appropriate.

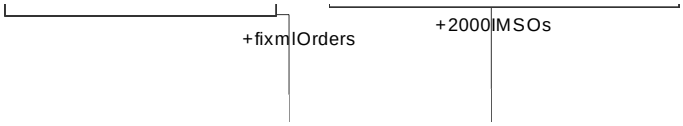
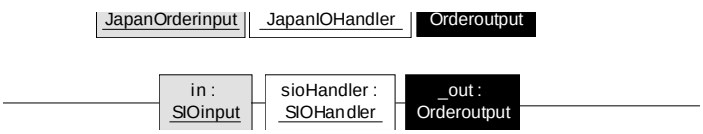

+fixmlOrders
+2000IMSOs

Figure 10-13 Pub/sub server - terminals

We are now ready to examine the workings of each of the


dardInternationalOrder) : Order
+sioHandler
.ransform(content : JapanOrder) : Order
+japanIOHandler


JapanOrderinput
JapanIOHandler
Orderoutput
in :
SIOinput
sioHandler :
SIOHandler
_out :
Orderoutput

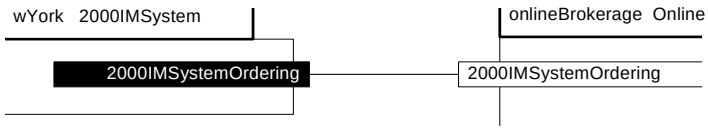
This explains why the notification output terminals of the
of its components. For each publisher, a combination of
source is used to generate subscriptions from some under

is required at this level.

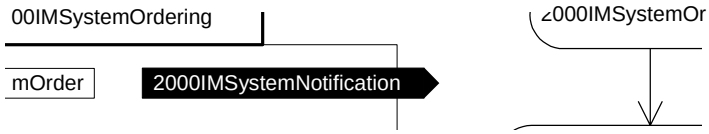
The back-end brokerage system is responsible for processi
processed and the order database is updated, this triggers


ownershipFilter :
AccountOwnership
in :
true :
ownershipAdder :
AddOwnership
__in :

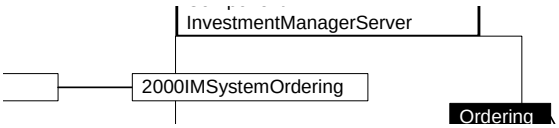
transformer. Although the second point mentions publish
subscription (a key part of the publication and subscription
this case. Rather, subscriptions are set up statically to filte

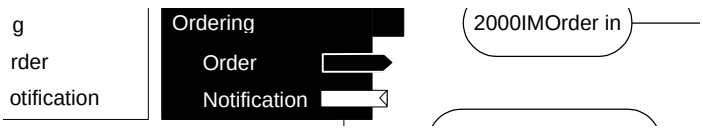

wYork 2000IMSystem
2000IMSystemOrdering
onlineBrokerage Online
2000IMSystemOrdering



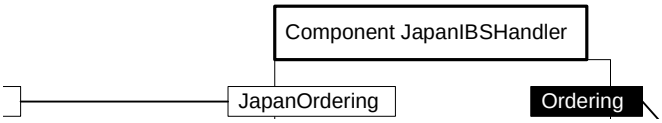

00IMSystemOrdering
mOrder
2000IMSystemNotification
2000IMSystemOr


apanOrdering
r
JapanNotification
JapanOrde


InvestmentManagerServer
2000IMSystemOrdering
Ordering


g
rder
otification
Ordering
Order
Notification
2000IMOrder in


Ordering
Order
Notification
r
ication
1990IMOrder in

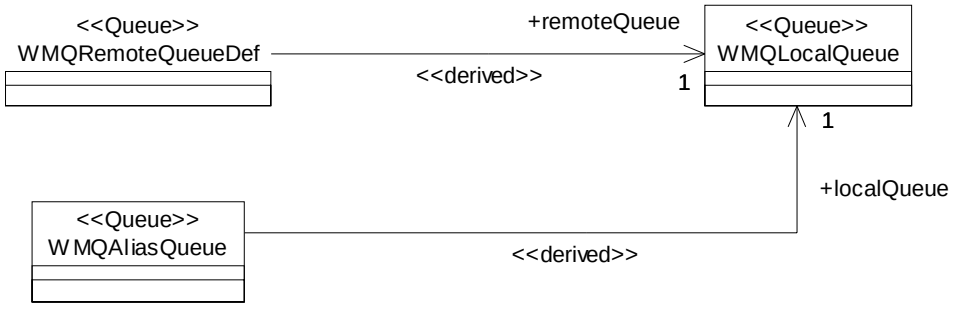

Component JapanIBSHandler
JapanOrdering
Ordering


ation
Ordering
Order
Notification
JapanOrder in


ing
Order
Notification
Ordering
Order
Notification
StandardIntlOrder in

which is unconstrained. The MQMD contains the fundam
allow efficient manipulation of a message by the WebSph
such as message expiry information and message identifie
portion of the message is effectively unconstrained, althou

ssage	ContentFormat	The WMQIControlMessage class is It allows control messages (such as subscription).


<<Queue>>
WMQRemoteQueueDef
+remoteQueue
<<Queue>>
WMQLocalQueue
<<derived>>
1
1
+localQueue
<<Queue>>
WMQAliasQueue
<<derived>>

level, processing is initiated by the receipt of a message o
WMQIInputNode. Consequently, an instance of WMQIM
least one WMQIInputNode. This (see Figure 12-3) has th
Output may be produced by one of three different node c

the compound node, this results in the emission of a mess
WMQIInputTerminalNode. Consequently, a WMQICompo
one WMQIInputTerminalNode. The results of message pr
WMQIOutputTerminals.

are specified in the IBM WebSphere MQ “Using the Con
document number SC34-5602). Each of these properties m
attribute of the appropriate type for each class.

. ilter	NA	.. Filter node routes a message accordi filter expression specified in SQL. The elements of the input message or mess data held in an external database. The

	Constraint	escription
	Input terminal is a QueuedInputTerminal. Expects to receive messages in	The WMQI message brok subscription operator.

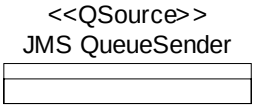
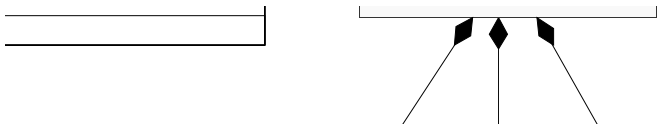
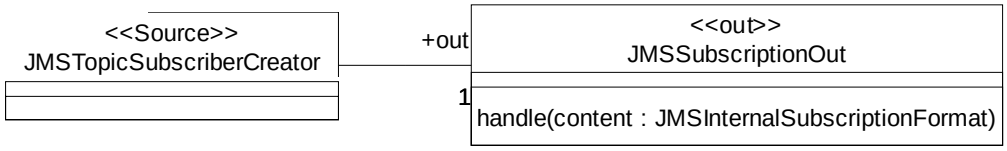


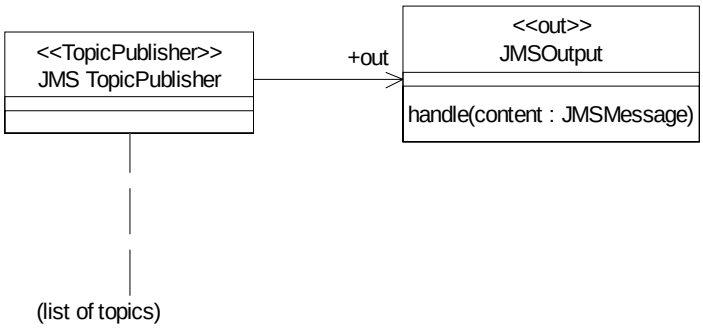
Figure 13-1 JMS QueueSender

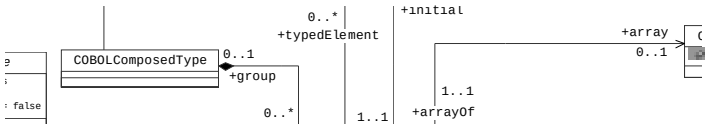


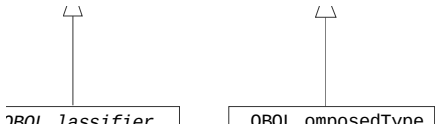
listening to the topic.

We model the ‘listener’ aspect as a class JMSSubscriberL that expects a JMSMessage as its input.


<<Source>>
JMSTopicSubscriberCreator
+out
1
<<out>>
JMSSubscriptionOut
handle(content : JMSInternalSubscriptionFormat)


<<TopicPublisher>>
JMS TopicPublisher
+out
<<out>>
JMSOutput
handle(content : JMSMessage)
(list of topics)


COBOLComposedType
0..1
+group
0..*
0..*
+typedElement
1..1
+initial
+array
0..1
1..1
+arrayOf
false



For example:

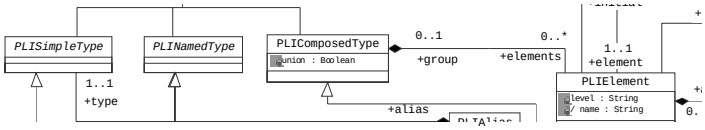
```
01 DATA-GROUP  PIC 9.
```

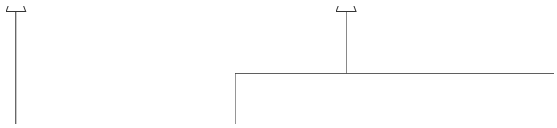
.7 *COBOLAlphaNumericType*

COBOLAlphaNumericType represents a picture string co
numeric characters.

COBOLInternalFloatType represents COBOL's internal fl

COBOLVariableLengthArray represents an array declared ON.


PLISimpleType
PLINamedType
PLIComposedType
union : Boolean
0..1
+group
0..*
+elements
1..1
+element
PLIElement
level : String
name : String
1..1
+type
+alias



PLIArray represents an n-dimensional collection of eleme
attributes.

14 PLIFixedBoundArray

PLIFixedBoundArray represents a fixed size array.

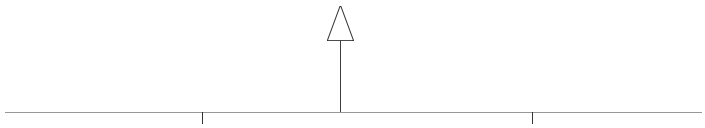
PLILabelType represents a label constant or the value of

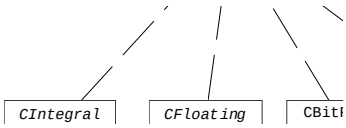
PLIPictureStringType represents a fixed-length character d
restriction that the data item can only contain characters f
complete set of available characters.

The C metamodel including C Main and User Types (i.e. u
Class instance at the M2 level.







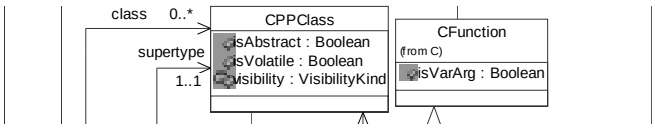

CIntegral
CFloating
CBitF

.4 CDatatype

CDatatype represents data types and native types.

14 CStructureContents

CStructureContents represent structured data types and st


class
0..*
CPPClass
isAbstract : Boolean
isVolatile : Boolean
visibility : VisibilityKind
supertype
1..1
CFunction
(from C)
isVarArg : Boolean

CPPExtern represents a function declared in a C program
C++ program. Declaring a function with the keyword 'ex
not to generate an internal name for the function. As a re


IMSTransactionMessage
OTMAPrefixFormat : OTMAPrefixFormats = one
StandardFieldsFlag : Boolean
<<enumeration>>
OTMAPrefixFormats
one


+StateData
+SecurityData

Figure A-5 OTMA Prefix - Security Data Defined Types

ControlData

ControlData is message-control information. It includes th message type, sequence numbers, flags and indicators.

Specifies how many segments are in the message. Thi
transaction and data message types, and it is mandator
messages.

- IMS OTMA messages without the OTMA prefix
- IMS basic messages to be sent to the application progr

- Utoken is VariableLengthField.

  Specifies the user token. The user ID and profile are u

- SynchronizationFlag is of type TsynchronizationFlag. I of the transaction. This flag controls and synchronizes client and server.

client, it is returned by the server in the output data m

TChainFlag

If the client connects first, it is notified by XCF when begins processing with a client-bid request.

- CBresynch (value X'0C') specifies a client-bid message

- REPresynch (value X'34') specifies the client's desired s
  A client sends the REPresynch command to IMS in re
  command received from IMS.

- UserData (value X'20') specifies that the message inclu the message prefix.
- ApplicationData (value X'10') specifies that the messag

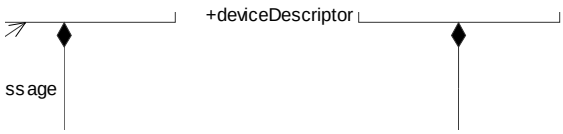
- If this flag is set for a command, IMS returns the arch application-data section of the ACK message. This fla commands /DISPLAY TRANSACTION and /DISPLA

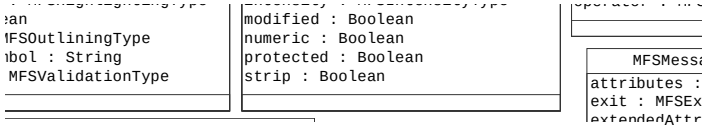
- Confirm (value X'01') specifies that synchronization is transaction output with the response flag set to Respon control information section of the message prefix.

business logic of the program.

Because the IMS application program I/O data structures
user interaction with these existing MFS applications, a w

MFSMessageField identifies the signature of an IMS FMS
both inputs and outputs. MFSMessageField associates wit
provides the linkage to the language and platform specific


+deviceDescriptor
ss age


ean
MFSOutliningType
bol : String
MFSValidationType
modified : Boolean
numeric : Boolean
protected : Boolean
strip : Boolean
MFSMessa
attributes :
exit : MFSEx

MFSDeviceDivision

This class encapsulates the MFS "DIV" statement.

attributes : MFSAttributeType

ATTR attribute.

PEN attribute.

Specifies a literal to be selected or an operator control fun

value : String

The default value of the device field.

referred to by a message output descriptor, the application
cursor position into this field as two binary halfwords con
respectively. Binary zeros in the named field cause the sp
cursor positioning during output. During input, binary zer

- ERASE
- FTAB

program can request autopaged output by setting the SCA
honored only if present in the first segment of the first LP

If a nonzero value is specified for byte 0, or for bit 6 or 7

specific features or hardware data stream dependencies.

Restriction: This keyword is optional and cannot be used specification for 3270 displays.

with an ampersand (&) designator character, the PEN= op specified in the MFLD statement.

SUB attribute.

Specifies the character used by MFS to replace any X'3F' stream. No translation occurs if this parameter is specified

MFSIfCondition

return an information message indicating that no next

- NEXTLP--NEXT LOGICAL PAGE specifies a reques the current message.

If ATTR=nn is specified, the minimum offset must be one
ATTR=2 is specified, pp must be at least 5, and, if ATTR
must be at least 7.

SOR attribute.

Specifies the source name of the FMT statement, which,

The value supplied for nn is the number of extended attrib
modified. The value of nn can be a number from 1 to 6. A
default to 1. Two additional bytes per attribute must be re

when it is invoked for this field. The value of exitnum ca
value of exitvect can range from 0 to 255. The address of
MFS editing, (but before NULL compression for option 1
exit routine, along with the vector defined for the field. (I

Corresponds to the 'literal' field in the following descripti

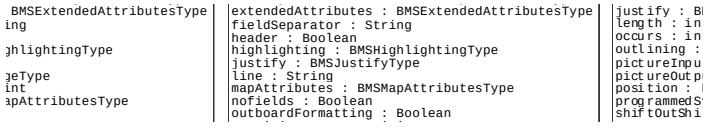
The device field name is specified via the 'deviceFields' r

(,SCA) defines this output field as the system control area
the output device. There can be only one such field in a l
must be in the first message segment of that page. If no lo

EDIT= parameter of the TRANSACT or NAME macro).
segment data is in non-graphic format (packed decimal, E
GRAPHIC=NO should be specified. When GRAPHIC=N
is invalid for MFLDs within this segment.

and MQ queues. The programmer creates an input file co
from the application to be displayed on output or formatte
'boilerplate' that should appear on the screen. Each field ca
for example, color, protection so that it cannot be overwr




BMSExtendedAttributesType
ing
ghlightingType
geType
int
apAttributesType
extendedAttributes : BMSExtendedAttributesType
fieldSeparator : String
header : Boolean
highlighting : BMSHighlightingType
justify : BMSJustifyType
line : String
mapAttributes : BMSMapAttributesType
nofields : Boolean
outboardFormatting : Boolean
justify : B
length : in
occurs : in
outlining :
pictureInpu
pictureOutp
position :
programmedS
shiftOutShi

ASKIP is the default and specifies that data cannot be key
the cursor to skip over the field.

BRT specifies that a high-intensity display of the field is

DET is specified.

FSET specifies that the modified data tag (MDT) for th

the DFHMSD macro.

If CTRL is used with cumulative BMS paging (that is, th
the BMS SEND MAP commands), it must be specified on

OCCURS operand cannot be specified.

The fields in a group must follow on; there can be gaps
fields from outside the group. A field name must be s

However, the validity of the input data is not checked
language when the map is used, so any desired checkin
application program. The length of the data associated

CURRENCY SIGN IS '$' WITH PICTURE
CURRENCY SIGN IS '£' WITH PICTURE

psid specifies a single EBCDIC character, or a hexadec
that identifies the set of programmed symbols to be us

The PS operand is ignored unless the terminal support

- C
- COBOL
- COBOL2

240, specifying the depth of a map as a number of line in the range 1-240, specifying the width of a map as a operand is required in the following cases:

entity.

- mode=OUT ¶ IN ¶ INOUT. OUT is default. INOUT sa processing. With IN, I is appended to mapname, with

For an assembler-language program, STORAGE=AUT
maps within a mapset are to occupy separate areas of s
one another.

BMSValidationType is the VALIDN statement. It specifies if the terminal supports it or this field can be processed b