CCA extracted from "A UML Profile for Enterprise Distributed Object Computing

Joint Final Submission

Part I

Version 1.0

Revised 22 August 2001

Submitted by:

CBOP
Data Access Technologies
DSTC
EDS
Fujitsu
IBM
Iona Technologies
Open-IT
Sun Microsystems
Unisys

Supported by:

Hitachi
SINTEF
NetAccount

OMG Document Number: ad/2001-08-19

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 sending email to issues@omg.org. Please reference precise page and section numbers, and state the specification name, version number, and revision date as they appear on the front page, along with a brief description of the problem. You will not receive any reply, but your report will be referred to the OMG Revision Task Force responsible for the maintenance of the specification. If you wish to be consulted or informed during the resolution of the submitted issue, indicate this in your email. Please note that issues appear eventually in the issues database, which is publicly accessible.

Contents

Figures

Tables

Chapter 1: Formal Response to the RFP

Table of Contents

1. Introduction

1.1 The Joint UML for EDOC Profile Submission

The Joint UML for EDOC Profile Submission is a specification for a UML Profile for Enterprise Distributed Object Computing (EDOC), prepared by the submitting team listed below in response to the OA&DTF RFP 6 (UML Profile for EDOC, OMG Document ad/99-03-10).

1.2 Co-submitting Companies

This submission is prepared by the following companies:

- CBOP
- Data Access Technologies
- DSTC
- EDS
- Fujitsu
- IBM
- Iona Technologies
- Open-IT
- Sun Microsystems
- Unisys

Supporting companies are:

- Hitachi
- Netaccount
- SINTEF

1.3 Status of this document

This document is a final revision to the Final Submission presented at the Danvers meeting of the OMG TC in July 2001. It contains some minor corrections of technical and editing errors and of formatting, but no substantial technical changes.

1.4 Guide to the Submission

1.4.1 Overall structure of the submission

This submission is divided into two parts as follows:

- Part I (this Part) is the normative specification of the UML Profile for EDOC;

- Part II contains a number of annexes which provide a set of non-normative mappings and a set of worked examples explaining the uses of the various parts of the Profile.

1.4.1.1 Part I

Part I contains six chapters as illustrated in Figure 1 below:

Chapter 1 is the formal response to the submission as required by the RFP.

Chapter 2 explains the overall rationale for the submission approach, and provides a framework for system specification using the EDOC Profile. It provides a detailed rationale for the modeling choices made and describes how the various elements in the submission may be used, within the viewpoint oriented framework of the Reference Model of Open Distributed Processing (RM-ODP), to model all phases of a software system's lifecycle, including, but not limited to:

- the analysis phase when the roles played by the system's components in the business it supports are defined and related to the business requirements;
- the design and implementation phases, when detailed specifications for the system's components are developed;
- the maintenance phase, when, after implementation, the system's structure or behavior is modified and tuned to meet the changing business environment in which it will work.

Chapter 3 is the Enterprise Collaboration Architecture (ECA) and contains the detailed profile specifications for platform/ technology independent modeling elements of the profile, specifically:

- the Component Collaboration Architecture (CCA) which details how the UML concepts of classes, collaborations and activity graphs can be used to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system;
- the Entities profile, which describes a set of UML extensions that may be used to model entity objects that are representations of concepts in the application problem domain and define them as composable components;
- the Events profile, which describes a set of UML extensions that may be used on their own, or in combination with the other EDOC elements, to model event driven systems;
- the Business Processes profile, which specializes the CCA, and describes a set of UML extensions that may be used on their own, or in combination with the other EDOC elements, to model workflow-style business processes in the context of the components and entities that model the business;
- the Relationships profile, which describes the extensions to the UML core facilities to meet the need for rigorous relationship specification in general and in business modeling and software modeling in particular.

Chapter 4 is the Patterns Profile, which defines how to use UML and relevant parts of the ECA profile to express object models such as Business Function Object Patterns (BFOP) using pattern application mechanisms.

Chapter 5 provides a set of technology specific mappings. It contains Java, Enterprise JavaBeans (EJB) and Flow Composition Model (FCM) metamodels abstracted from their respective specifications.

- The EJB metamodel is intended to provide sufficient detail to support the creation assembly and deployment of Enterprise JavaBeans.
- The Java metamodel is intended to provide sufficient detail to support the EJB metamodel.
- The Flow Composition Model provides a common set of design abstractions across a variety of flow model types used in message brokering and delivery.

Chapter 6 (UML Profile for MOF) is a normative two way mapping between UML and the MOF. Although this is not called for in the RFP, it is deemed essential, since, for the profiles proposed to be understood, it has been necessary to include metamodels that explain the concepts that the profiles express.

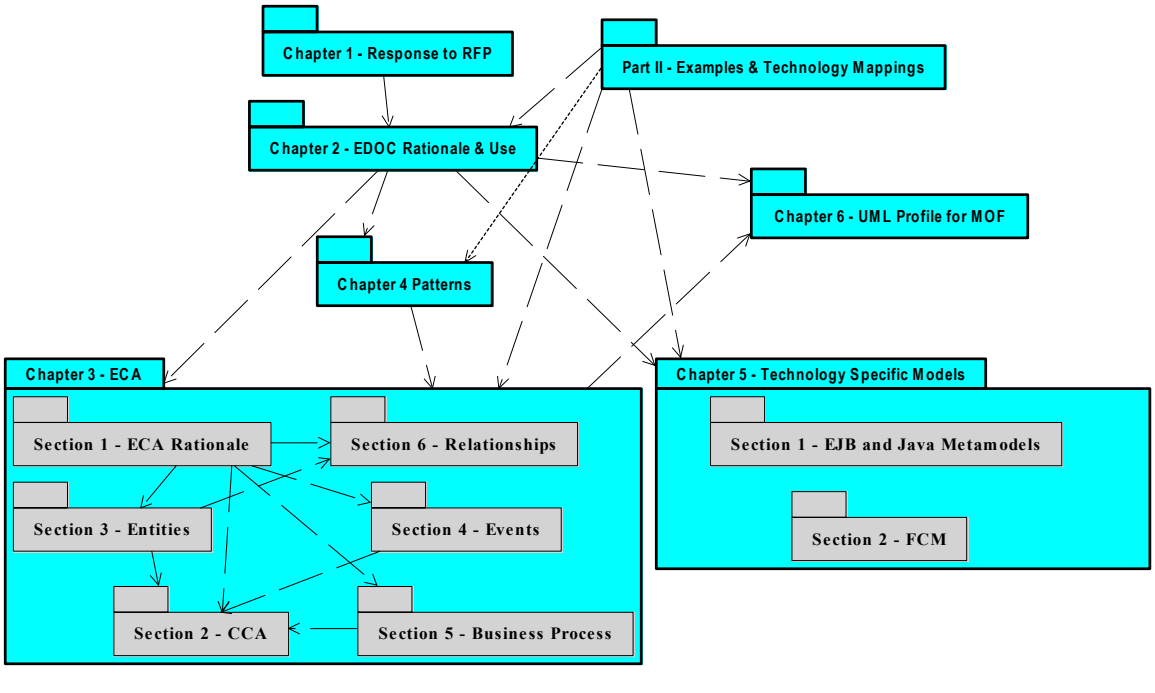

Figure 1: UML for EDOC Submission Structure

1.4.1.2 Part II

Part II of this submission, (ad/2001/08/20) is non-normative and contains supporting information in the form of the following Annexes:

- Annex A - Procurement, Buyer/Seller example
- Annex B - Meeting Room example
- Annex C - Hospital example
- Annex D - Examples of Patterns
- Annex E - Technology mappings from EDOC to Distributed Component and Message Flow Platform Specific Models

In addition, XMI and DTD data files for the metamodels in the EJB/Java/FCM profiles are

included in the zip file containing this Part II of the submission, in the folder named “XMI and DTDs”.

1.4.2 Structure of Chapter 1

Section 1 provides contact information and a guide to this submission.

Section 2 is the proof of concept statement.

Section 3 explains how this submission satisfies the mandatory requirements of the RFP.

Section 4 summarizes the rationale for the approach taken in this submission (which is described in detail in Part II).

Section 5 provides a statement of Conformance Points for this specification.

Section 6 discusses changes to OMG adopted standards.

A Glossary and a List of References are provided at the end of this Part.

1.5 Missing Items

None

1.6 Submission contact points

1.6.1 CBOP

Akira Tanaka
Hitachi, Ltd.,
Software Division, Enterprise Business Planning, Product Planning Dept.,
5030 Totsuka-cho, Totsuka-ku, Yokohama 244-8555, Japan
e-mail: tanakaak@soft.hitachi.co.jp
phone: +81(45)862-8735
fax:+81(45)865-9020

Hajime Horiuchi
Tokyo International University
1-13-1 Matoba-kita, Kawagoe-shi, Saitama 350-1102, Japan
Phone: +81-492-32-1111
Email:hori@tiu.ac.jp

Marika Iizuka
Technologic Arts Inc.
Cosmos Hongo Bld. 9F, 4-1-4 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
Phone: +81-3-5803-2788
Email: marika@tech-arts.co.jp

Masaharu Obayashi
Kanrikogaku Ltd.
Meguro Suda Bldg., 3-9-1 Meguro, Meguro-ku, Tokyo 153-0063, Japan
Phone: +81-3-3716-6300
Email: obayashi@kthree.co.jp

Yoshihide Nagase
Technologic Arts Inc.
Cosmos Hongo Bld. 9F, 4-1-4 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
Phone: +81-3-5803-2788
Email: yoshi@tech-arts.co.jp

1.6.2 Data Access Technologies

Cory B. Casanave and Antonio Carrasco-Valero
14000 SW 119 Av., Miami, FL 33186, USA
Phone: +1 305 234 7077
Email: cory-c@ enterprise-component.com , antonio-c@ enterprise-component.com

1.6.3 DSTC

Mr. Keith Duddy,
CRC for Enterprise Distributed Systems Technology (DSTC)
University of Queensland
Brisbane 4072
Australia
Phone: +61 7 3365 4310
Fax: +61 7 3365 4311
Email:dud@dstc.edu.au, edoc-rfp1@dstc.edu.au
WWW: www.dstc.edu.au

1.6.4 EDS

Fred Cummins
EDS
5555 New King St., MS 402, Troy, MI 48098, USA
Phone: (248) 696-2016
Email: fred.cummins@eds.com

1.6.5 Fujitsu

Mr Hiroshi Miyazaki
Fujitsu Limited
1-9-3, Nakase Mihama-ku, Chiba-shi, Chiba 261-8588, Japan
Phone: +81 43 299 3531 ext 4669
e-mail: <miyazaki@tokyo.se.fujitsu.co.jp>

1.6.6 IBM

Stephen A. Brodsky, Ph.D.
International Business Machines Corporation
555 Bailey Ave., J8RA/F320
San Jose, CA 95141
Phone: +1 408 463 5659
E-mail: SBrodsky@us.ibm.com

1.6.7 Iona

David Frankel
Iona Technologies
10 North Church Street, West Chester, Pa 19380, USA
Phone: 610 429 1553
Email: david.frankel@iona.com

1.6.8 Open-IT

Mr Sandy Tyndale-Biscoe
Open-IT Ltd
Cedarcroft, Sunny Way, Bosham, CHICHESTER, West Sussex, PO18 8HQ, U.K.
Phone: +44 (0)1243 57 22 23
e-mail: <sandy@open-it.co.uk>

1.6.9 SINTEF

Dr. Arne J. Berre
SINTEF Telecom and Informatics
Forskningsveien 1, Blindern, 0314 Oslo, Norway
Phone: +47 22 06 74 52
e-mail: Arne.J.Berre@informatics.sintef.no

1.6.10 Sun Microsystems

Karsten Riemer,
b2b Architect, XML technology Center,
Sun Microsystems, Inc., Burlington, MA 01803, USA
Phone 781-442-2679
e-mail karsten.riemer@sun.com

1.6.11 Unisys

Sridhar Iyengar
Unisys Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
Phone: +1 949 380 5692
E-mail: sridhar.iyengar2@unisys.com

2. Proof of Concept

This submission is a practical approach to the need for specifying EDOC systems, based on the following real world experience of the companies concerned:

2.1 CBOP

CBOP is a consortium in Japan, promoting the reuse and the sharing of business domain models and software components. The submission of the pattern mechanism to the UML profile for EDOC RFP was based on the CBOP standards that are focused on the normalization of business object patterns for modeling. Current work of CBOP is, *inter alia*, concerned with the development of UML tools that enable the application of patterns in object modeling with UML. The EDOC standard will be taken in to account in these tools as well as the CBOP standards.

2.2 Data Access Technologies

The CCA profile (Chapter 3 Section 2) is based on product development done by Data Access Technologies under a cooperative agreement with the National Institute of Technologies - Advanced Technology Program. The basis for CCA has been proven in two related works - one as a distributed user interface toolkit for Enterprise Java Beans and more recently as the basis for "Component X Studio" which provides drag-and-drop assembly of server-side application components. Component-X Studio is has been released as a product. Portions of this same model have also been incorporated into ebXml for its specification schema, giving CCA an XML based technology mapping. Finally, portions of CCA and the related entity model derive from standards, development and consulting work done in relation to the "Business Object Component Architecture" which, while never standardized has proven to be a solid foundation for modeling and implementing a systems information viewpoint. In all cases of the above works, model based development has been used throughout the lifecycle, from design to deployment - proving the sufficiency of the base models to drive execution.

2.3 DSTC

DSTC has used its dMOF product to develop a MOF respository and Human Usable Textual Notation I/O tools which support modeling of Business Processes conforming to the metamodel in Chapter 3, Section 6 (Business Process profile). Significant Business Process models have been created using these generated tools, and mapped using XSLT into XML workflow process definitions, which execute on the DSTC's Breeze workflow engine. dMOF is a commercial product installed at many customer sites world-wide, and Breeze is in development and is currently being beta-tested by four DSTC partner organizations.

In addition the dMOF tool has been used to validate the MOF conformance of all the meta-models in Chapter 3. XMI documents containing these meta-models will be submitted as separate conveniece documents.

2.4 EDS

EDS developed the Enterprise Business Object Facility (EBOF) product in conjunction with work on the Business Object Facility specification. This product serves as a proof of concept for important aspects of this submission. It incorporated UML models as the basis for generating executable, distributed, CORBA applications. This involved consideration of transactions, persistence, management of relationships, operations on extents, performance optimization and many other factors. This product was sold to a major software vendor.

2.5 Fujitsu

This submission is based in part upon Fujitsu's system analysis and design methodology, "Application Architecture/Business Rule Modeling". The methodology is built into Fujitsu's product, "Application Architecture / Business Rule Modeler - AA/BRMODELER", which has been used for the development of many mission critical business systems. Although applied mainly to the development of COBOL applications, the methodology includes object-oriented characteristics. In this submission, the elements of the methodology and its related product are represented as UML elements and extensions. In the methodology, the specification of business rules is of special concern. The business rules are separated in types and attributed to objects corresponding to the types. These rules are represented in a formal grammar, and they are compiled into executable programs by using AA/BRMODELER. AA/BRMODELER has sold approximately 5000 sets in Japan since it was developed in 1994. It has been applied to approximately 300 projects, some of scale greater than 7,000 person-months.

2.6 IBM

IBM has extensive experience in enterprise architectures, Java, Enterprise Java Beans, CORBA, UML, MOF, and metadata. The WebSphere, MQ, and VisualAge product lines provide sophisticated analysis, design, deployment, and execution functionality embodying all of the key representative technologies.

2.7 Iona

The Relationships Profile is based on many years of modeling experience in industry and in the development of related products and standards. It uses ISO's General Relationship Model and the work of Haim Kilov and James Ross in their book "Information Modeling", which is based on long-term modeling experience in areas such as telecommunications, finance, insurance, document management, and business process change.

The Process Profile incorporates Iona experience modeling enterprise processes with customers from use case descriptions, business models, and other IT system requirements information. It is also based on experience developing process definition and management products for environments ranging from concurrent engineering to document processing.

2.8 Open-IT and SINTEF

The profile incorporates results and experience from the UML profile and associated lexical language that was developed in the European Union funded OBOE project. As part of this project supporting tools were developed and the technology was applied at a user site . A full description of the project is available at [7].

The ODP concepts have been applied for the development of the OMG Finance domain General Ledgers specification in the COMPASS project, and a mapping framework for Microsoft COM has been developed by Netaccount (formerly Economica). More information on this is available at [6].

The ODP concepts have also been applied in the domain of geographic information systems. The DISGIS project has demonstrated the usefulness of the separation of concerns in terms of the 5 viewpoints defined by the RM-ODP, and developed an interoperability framework based on this (See [5]). The use of the ODP viewpoints have also been found useful in the context of geographic information system standardization in ISO/TC211 (See [8]) and the Open Geodata Consortium (See [9]).

The enterprise specification concepts have been derived from work for the UK Ministry of Defence and Eurocontrol together with participation in the development of the ODP – Enterprise Language standard [4].

2.9 Sun Microsystems

Sun Microsystems' internal IT group has successfully implemented large scale Enterprise Integration using a conceptual meta-model close to that defined in the Events profile (Chapter 3 Section 4), covering business process, entity, and event architecture. While this has not been using UML, the work modeled the enterprise and the interaction between system components based on an enterprise business object/event information model. Business objects and events have been modeled in a Sun IT internal language, SDDL, a self describing data language, the syntax of which is equivalent to the modeling framework proposed here.

This implementation is successful, and by a rough estimate 50% of Sun's key applications participate in event driven processes, and in total about a million event notifications are sent among these applications every day.

2.10 Unisys

Unisys has extensive experience in enterprise architectures, commercial metadata repositories, metadata interchange, Java, Enterprise Java Beans, CORBA, COM+, UML, and MOF. Unisys products provide extensive and distributed metadata management services. Unisys has designed numerous metamodels using UML, and has deployed numerous metamodels using MOF, including metamodels of Java, CORBA IDL, UML, and CWM.

2.11 ebXML

The ebXML Business Process Specification Schema (BPSS), which was adopted as a specification on May 11th 2001, is aligned with and validates the Component Collaboration Architecture (CCA). This alignment was demonstrated as part of the ebXML "proof of concept" on the same day. This alignment validates the use of CCA concepts to express Business-to-Business processes in a precise (executable) manner. The United Nations and Oasis jointly sponsor EbXML.

3. Response to RFP Requirements

3.1 General Mandatory Requirements

The proposal addresses those mandatory General Requirements on Proposals (see UML Profile for EDOC, OMG Document ad/99-03-10, section 5.1) which are relevant to it. Specifically, the proposal:

- is precise and functionally complete, and has no implied or hidden interfaces, operations, or functions required to enable an implementation of the proposed specification (5.1.3);
- clearly distinguishes mandatory and optional specification elements (5.1.4);
- makes use of the existing UML specification and does not specify any changes or heavyweight extensions to it (5.1.5 and 5.1.6);
- factors out into separate Chapters functions that can be used in different contexts (5.1.7);
- preserves the implementation flexibility of the UML specification on which it is based (5.1.11);
- does not impact the interoperability of independent UML implementations (5.1.12);
- has compatibility with the architecture for system distribution defined in ISO/IEC 10746, Reference Model of Open Distributed Processing (ODP) ([1], [2], [3]) as an important objective: Section 3 of Chapter 2 describes how the concepts and profile elements defined in Chapters 3, 4 and 5 can be used to develop a full set of specifications of an EDOC system that takes as a framework the separation of concerns as defined by RM-ODP viewpoints (see [3]) (5.1.13).

3.2 Specific Mandatory Requirements

3.2.1 Component Modeling

Components are modeled using the CCA profile (Chapter 3 Section 2). The following characteristics are covered as described:

- **Transactional characteristics:** requirements for Transactional characteristics are specified as characteristics of a Port (CCA profile (Chapter 3 Section 2)).
- **Security characteristics and details of the security services employed (such as authentication, authorization, message protection, data protection, security logging, and non-repudiation):** this submission provides no specific modeling mechanisms for expressing security characteristics and details of the security services employed.
- **Persistence characteristics and details of interaction with persistent stores:** CCA Process Components (CCA profile (Chapter 3 Section 2)) may be specified as persistent as can Identifiable Entities and Processes (Entities profile (Chapter 3 Section 3).

- **Packaging and deployment characteristics:** A ComposedComponent (CCA profile (Chapter 3 Section 2)) can describe a logical package that is independently deployable.

Chapter 5 describes mappings to two widely used industry component model architectures, EJB and FCM.

3.2.2 Modeling of Business Process, Entity, Rule, and Event Objects

Business Processes are modeled using respectively, the BusinessProcess, CompoundTask and Activity stereotypes (Business Process profile, (Chapter 3 Section 5)) for the enterprise viewpoint and the Process Component stereotype (CCA, Chapter 3 Section 2), for the computational viewpoint.

Business Entities are modeled in the computational and information viewpoints primarily using the concepts defined in the Entities profile (Chapter 3 Section 3), particularly the stereotype Entity. These are bound to instances of the ProcessRole stereotype from the Business Process Profile in the enterprise viewpoint.

Business Rule objects may be modeled using either the Events profile (BusinessRule) or, where they apply only to entities, the Entities profile (Rule). Selection and Creation Rules for the binding of ProcessRoles are modeled in the Business Process Profile.

Events may be modeled using the BusinessEvent, EntityEvent or ProcessEvent stereotypes from the Events Profile (for the computational viewpoint and, occasionally, for the enterprise viewpoint).

3.2.3 Specification of Business Process Objects

The definition of Business Processes and associated Business Rules in the enterprise specification (using the Business Process profile, (Chapter 3 Section 5)) provides a definition of the constituent activities of those processes enacted by ProcessComponents identified in the computational specification (using the CCA profile (Chapter 3 Section 2)). The detailed specification of temporal and data dependencies between activities in a Business Process is also defined in the Business Process Profile, while the initiation of business process objects at runtime is provided by the computational specification using the CCA profile. It is recognized that the specification of Business Process Objects may be related to the OMG Workflow Management Facility.

The Entities profile provides the linkage between CCA, Entities and Processes (business process objects). The process component is essentially a process object (containing other components). The Process Profile describes specializations of Process Components and their usages consistent with the OMG workflow specification. Business Processes can be seen either as objects with interfaces to be invoked, or as containers for the context data of a process and managers of the the activities whose execution ordering they define. The activities in turn use other Process Components to do their work. The Process Profile can be considered to be a particular process paradigm; there are others.

3.2.4 Specification of Relationships

This submission provides mechanisms for the specification of additional, specialized relationship semantics beyond the base UML metamodel as follows:

- **Additional properties of relationships to specify constraints or operational semantics:** these are described in the Relationships profile (Chapter 3 Section 6).
- **Classifications of relationships by their properties:** these are described in the Relationships profile (Chapter 3 Section 6).
- **Derivation of pre and post conditions for create/read/update/delete ("CRUD") operations applied to participants in the relationships, based on the above properties and classifications:** these are described in the Entities profile (Chapter 3 Section 3), and are implemented using the Relationships profile (Chapter 3 Section 6)

3.2.5 Meta-Object Facility Alignment

This submission specifies a UML profile as defined in UML 1.4 and a set of MOF models that are isomorphic to the profile. In addition, Chapter 6 defines a UML Profile for MOF, which standardizes the way in which UML is used to represent MOF models. To date the only standard printable representation of MOF models was defined by XMI. Not only are all the MOF models which express the relationships between the EDOC modeling concepts conformant to MOF 1.3, but they are also represented in diagrammatic form in this submission using the UML Profile for MOF.

Each sub-profile of this submission expresses the relevant set of concepts of EDOC using both a MOF model which has no dependencies on the UML metamodel, as well as a Profile of UML which specializes UML modeling concepts to produce the EDOC semantics. In addition the correspondences between the MOF metamodel elements and the Profile package model elements is explained.

The provision of MOF models separate from but corresponding to the UML Profile has many benefits:

- The EDOC concepts are complex and are not easily explained or understood using only a UML Profile.
- The EDOC concepts, when explained using only MOF classes, attributes and associations, form a relatively small set of model elements that are directly related to one another, and may be easily depicted graphically without the need to expose derived meta-associations and meta-attributes.
- The MOF models form a repository and model interchange basis for EDOC designs which do not require tool vendors to implement the large part of UML which is being profiled. In addition, the XMI generated from the MOF models will allow interchange of EDOC designs which UML tools expressing EDOC designs in terms of stereotypes and tagged values, will be incapable of exchanging using XMI for UML. (The UML "Physical Metamodel" defines MOF meta-classes for exchanging Profiles, but not for exchanging models that are conformant to a particular Profile.)

3.2.6 Proof of Concept of Profile

Examples of the use of the profiles are Provided in Part II of this submission.

3.2.7 Proof of Concept of Mappability

A set of non-normative mappings from the ECA Profile to various technologies, including CORBA Workflow Management Facility, is provided in the Annexes, Part II of this submission.

3.3 Optional Requirements

There are none.

3.4 Subset Integrity

There are no dependencies outside the specified subset of UML.

3.5 Simplification of and Aid to the Development Process

The primary sense in which use of the EDOC Profile simplifies and aids the development process is that it meets the requirements of the RFP. As stated in the RFP, "successful implementation of an enterprise computing system requires the operation of the system to be directly related to the business processes it supports. A good object-oriented model for an enterprise computing system must therefore provide a clear connection back to the business processes and business domain that are the basis for the requirements of the system. However, this model must also be carried forward into an effective implementation architecture for the system. This is not trivial because of the demanding nature of the target enterprise distributed computing environment."[1].

This submission provides a set of standard ways to use the UML to produce a set of linked and traceable models of a software system, which are applicable to all phases of that system's lifecycle, including, but not limited to:

- the analysis phase when the roles played by the system's components in the business it supports are defined and related to the business requirements;
- the design and implementation phases, when detailed specifications for the system's components are developed;
- the maintenance phase, when, after implementation, the system's behavior is modified and tuned to meet the changing business environment in which it will work.

The use of such a standard set of modeling techniques will considerably aid the software development process by:

- providing more rigorous linkages between the various development phases (concept, elaboration, construction and transition);
- reducing variability in modeling techniques that lead to misunderstandings between team members, and hence re-work;
- allowing more precise specification of software components and hence more opportunity for re-use;
- allowing more precise specification of a system's role in the business it supports, thereby reducing user dissatisfaction and re-work.

3.6 Tool support

The EDOC Profile is entirely conformant with the UML metamodel, in that it makes no heavyweight extensions to that metamodel. Thus, in theory, any tool that is fully compliant with UML1.4 can implement the profile. However, not many tools *fully* implement the UML1.4 metamodel. Therefore, the profile generally only uses the commonly used UML constructs, and, provided the tool concerned implements the UML extensions mechanism

[1] RFP p19 under the heading of "Enterprise Computing Systems"

fully, there should be no difficulty in it supporting the metamodels incorporated in this profile.

In addition the EDOC concepts represented by the Profile are modeled using MOF 1.3 metamodels, and may therefore form the basis for tools that are EDOC-specific, and do not implement UML. The use of MOF conformant metamodels imply that a MOF IDL repository, and XMI interchange DTD may be automatically generated from this specification to assist tool vendors wishing to create EDOC-specific tools.

This profile also makes some recommendations about notation. Use of these recommendations is optional but it would considerably enhance communication and comprehension of EDOC models.

3.7 Alignment with Action Semantics for UML

The submitters' view is that Action Semantics for UML describes what happens inside an object, whereas the EDOC Profile provides a modeling framework for describing how objects are used to implement enterprise systems.

4. Conformance Issues

4.1 Summary of optional versus mandatory interfaces

For a modeling tool to claim compliance to the EDOC specification it must implement at least one of the mandatory compliance points in Section 4.2.1, and state the name of the compliance point(s). The mandatory compliance points are all variations on the ability to model or interchange designs using the Enterprise Component Architecture (ECA), which forms the core of EDOC.

There are a number of other normative profiles and metamodels contained within this specification, and these are given named optional compliance points in Section 4.2.2.

4.2 Proposed Compliance Points

4.2.1 Mandatory Compliance Points

At least one of the following compliance points must be implemented for a tool or model to claim compliance with the EDOC specification:

Mandatory Compliance Point Name	**MOF Repository**	**MOF XMI interchange**	**UML Profile**	**UML Profile XMI interchange**
ECA MOF Repository	yes	no	no	no

Mandatory Compliance Point Name	MOF Repository	MOF XMI interchange	UML Profile	UML Profile XMI interchange
ECA MOF XMI Interchange	no	yes	no	no
ECA MOF Repository and Interchange	yes	yes	no	no
ECA UML Profile	no	no	yes	no
ECA UML XMI Interchange	no	no	no	yes
ECA UML Profile and Interchange	no	no	yes	yes

Table 1: Mandatory Compliance Points

The columns in Table 1 are defined as follows:

4.2.1.1 MOF Repository

Any implementation of a CORBA server defined by generating and implementing the IDL and its semantics, as defined in MOF 1.3 (formal/00-04-03), from MOF models defined in the package "ECA" and all of its sub-packages.

4.2.1.2 MOF XMI interchange

Any implementation of a service that produces XML documents that conform to the XMI DTD produced by applying the XMI 1.1 specification (formal/00-11-02) to the MOF package "ECA" and all of its sub-packages.

4.2.1.3 UML Profile

Any tool or model that implements the Profile mechanisms defined in UML 1.4 (ad/01-02-13), and which is populated with stereotypes, tagged values and constraints defined in the ECA «profile» Package, and all of its sub-packages, and provides standard UML1.4 notation for such models.

4.2.1.4 UML Profile XMI interchange

Any tool or model which is capable of producing XML documents that comform to the XMI DTD produced by applying the XMI 1.1 specification (formal/00-11-02) to the MOF package UML Interchange metamodel, as defined in chapter 5 of UML 1.4 (ad/01-02-13), and correctly encodes the stereotypes and tagged values defined in the ECA «profile» Package, and all of its sub-packages.

4.2.2 Optional Compliance Points

The submission has the following optional compliance points:

4.2.2.1 Patterns Profile

Any tool that implements the Profile mechanisms defined in UML 1.4 (ad/01-02-13), and which is populated with stereotypes, tagged values and constraints defined in the EDOC::Pattern «profile» Package, and all of its sub-packages.

4.2.2.2 Patterns Model

Or any tool that implements the semantics of the MOF metamodel EDOC::Pattern package (Chapter 4), and allows access to patterns generated either by generated MOF 1.3 (formal/00-04-03) IDL interfaces or via XML documents produced via the application of XMI 1.1 (formal/00-11-02) to the metamodel.

4.2.2.3 Java Model

Use of the normative Java metamodel (Chapter 5, section 1.1) by instantiation, code generation, invocation, or serialization as defined by the MOF 1.3 (formal/00-04-03) and XMI 1.1 (formal/00-11-02) specifications.

4.2.2.4 EJB Model

Use of the normative EJB metamodel (Chapter 5, Section 1.2) by instantiation, code generation, invocation, or serialization as defined by the MOF 1.3 (formal/00-04-03) and XMI 1.1 (formal/00-11-02) specifications.

4.2.2.5 FCM Model

Use of the normative FCM metamodel (Chapter 5, Section 2) by instantiation, code generation, invocation, or serialization as defined by the MOF 1.3 (formal/00-04-03) and XMI 1.1 (formal/00-11-02) specifications.

4.2.2.6 UML Profile for MOF

Any tool that implements the Profile mechanisms defined in UML 1.4 (ad/01-02-13), and which is populated with stereotypes, tagged values and constraints defined in the uml2mof «profile» Package (Chapter 6).

4.2.2.7 CCA Notation

Any tool or model which implements the CCA notation as specified in Chapter 2, Section 2.3.

4.2.2.8 Business Process Notation

Any tool or model which implements the business process notation as specified in Chapter 2, Section 5.4.

5. Changes or extensions required to adopted OMG specifications

No changes of extensions to adopted OMG specifications are required for the adoption of this submission.

6. Proof of Concept mappings

The proof of concept mappings can be found in Part II of this Submission.

Chapter 2: EDOC Profile – Rationale and Application

Table of Contents

2. The Component Collaboration Architecture

The Component Collaboration Architecture (CCA) details how the UML concepts of classes, collaborations and activity graphs can be used to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system.

2.1 Rationale

2.1.1 Problems to be solved

The information system has become the backbone of the modern enterprise. Within the enterprise, business processes are instrumented with applications, workflow systems, web portals and productivity tools that are necessary for the business to function.

While the enterprise has become more dependent on the information system the rate of change in business has increased, making it imperative that the information system keeps pace with and facilitates the changing needs of the enterprise.

Enterprise information systems are, by their very nature, large and complex. Many of these systems have evolved over years in such a way that they are not well understood, do not integrate and are fragile. The result is that the business may become dependent on an information infrastructure that cannot evolve at the pace required to support business goals.

The way in which to design, build, integrate and maintain information systems that are flexible, reusable, resilient and scalable is now becoming well understood but not well supported. The CCA is one of a number of the elements required to address these needs by supporting a scalable and resilient architecture.

The following subsections detail some of the specific problems addressed by CCA.

2.1.1.1 Recursive decomposition and assembly

Information systems are, by their very nature, complex. The only viable way to manage and isolate this complexity is to decompose these systems into simpler parts that work together in well-defined ways and may evolve independently over time. These parts can than be separately managed and understood. We must also avoid re-inventing parts that have already been produced, by reusing knowledge and functionality whenever practical.

The requirements to decompose and reuse are two aspects of the same problem. A complex system may be decomposed “top down”, revealing the underlying parts. However, systems will also be assembled from existing or bought-in parts – building up from parts to larger systems.

Virtually every project involves both top-down decomposition in specification and “bottom up” assembly of existing parts. Bringing together top-down specification and

bottom-up assembly is the challenge of information system engineering.

This pattern of combining decomposition in specification and assembly of parts in implementation is repeated at many levels. The composition of parts at one level is the part at the next level up. In today's web-integrated world this pattern repeats up to the global information system that is the Internet and extends down into the technology components that make up a system infrastructure – such as operating systems, communications, DBMS systems and desktop tools.

Having a rigorous and consistent way to understand and deal with this hierarchy of parts and compositions, how they work and interact at each level and how one level relates to the next, is absolutely necessary for achieve the business goals of a flexible and scalable information systems.

2.1.1.2 Traceability

The development process not only extends "up and down" as described above, but also evolves over time and at different levels of abstraction. The artifacts of the development process at the beginning of a project may be general and "fuzzy" requirements that, as the project progresses, become precisely defined either in terms of formal requirements or the parts of the resulting system. Requirements at various stages of the project result in designs, implementations and running systems (at least when everything goes well!). Since parts evolve over time at multiple levels and at differing rates it can become almost impossible to keep track of what happened and why.

Old approaches to this problem required locking-down each level of the process in a "waterfall". Such approaches would work in environments where everything is known, well understood and stable. Unfortunately such environments seldom, if ever, occur in reality. In most cases the system becomes understood as it evolves, the technology changes, and new business requirements are introduced for good and valid reasons. Change is reality.

Dealing with this dynamic environment while maintaining control requires that the parts of the system and the artifacts of the development process be traceable both in terms of cause-effect and of changes over time. Moreover, this traceability must take into account the fact that changes happen at different rates with different parts of the system, further complicating the relationships among them. The tools and techniques of the development process must maintain and support this traceability.

2.1.1.3 Automating the development process

In the early days of any complex and specialized new technology, there are "gurus" able to cope with it. However, as a technology progresses the ways to use it for common needs becomes better understood and better supported. Eventually those things that required the gurus can be done by "normal people" or at least as part of repeatable "factory" processes. As the technology progresses, the gurus are needed to solve new and harder problems – but not those already solved.

Software technology is undergoing this evolution. The initial advances in automated software production came from compilers and languages, leading to DBMS systems, spreadsheets, word processors, workflow systems and a host of other tools. The end-user today is able to accomplish some things that would have challenged the gurus of 30 years ago.

This evolution in automation has not gone far enough. It is still common to re-invent infrastructures, techniques and capabilities every time a new application is produced. This is not only expensive, it makes the resulting solutions very specialized, and hard to integrate and evolve.

Automation depends on the ability to abstract away from common features, services, patterns and technology bindings so that application developers can focus on application problems. In this way the ability to automate is coupled with the ability to define abstract viewpoints of a system – some of which may be constant across the entire system.

The challenge today is to take the advances in high-level modeling, design and specification and use them to produce factory-like automation of enterprise systems. We can use techniques that have been successful in the past, both in software and other disciplines to automate the steps of going from design to deployment of enterprise scale systems. Automating the development process at this level will embrace two central concepts; reusable parts, and model-based development. It will allow tools to apply pre-established implementation patterns to known modeling patterns. CCA defines one such modeling pattern.

2.1.1.4 Loose coupling

Systems that are constructed from parts and must survive over time, and survive reuse in multiple environments, present some special requirements. The way in which the parts interact must be precisely understood so that they can work together, yet they must also be loosely coupled so that each may evolve independently. These seemingly contradictory goals depend on being able to describe what is important about how parts interact while specifically not coupling that description to things that will change or how the parts carry out their responsibility.

Software parts interact within the context of some agreement or contract – there must be some common basis for communication. The richer the basis of communication the richer the potential for interaction and collaboration. The technology of interaction is generally taken care of by communications and middleware while the semantics of interaction are better described by UML and the CCA.

So while the contract for interaction is required, factors such as implementation, location and technology should be separately specified. This allows the contract of interaction to survive the inevitable changes in requirements, technologies and systems.

Loose coupling is necessarily achieved by the capability of the systems to provide “late binding” of interactions to implementation.

2.1.1.5 Technology Independence

A factor in loose coupling is technology independence i.e. the ability to separate the high-level design of a part or a composition of parts from the technology choices that realize it. Since technology is so transient and variations so prevalent it is common for the same “logical” part to use different technologies over time and interact with different technologies at the same time. Thus a key ingredient is the separation high-level design from the technology that implements it. This separation is also key to the goal of automated development.

2.1.1.6 Enabling a business component Marketplace

The demand to rapidly deploy and evolve large scale applications on the internet has made brute force methods of producing applications a threat to the enterprise. Only by being able to provision solutions quickly and integrate those solutions with existing legacy applications can the enterprise hope to achieve new business initiatives in the timeframe required to compete.

Component technologies have already been a success in desktop systems and user interfaces. But this does not solve the enterprise problem. Recently the methods and technologies for enterprise scale components have started to become available. These include the "alphabet soup" of middleware such as XML, CORBA, Soap, Java, ebXml, EJB & .net., What has not emerged is the way to bring these technologies together into a coherent enterprise solution and component marketplace.

Our vision is one of a **simple** drag and drop environment for the **assembly** of **enterprise components** that is integrated with and leverages **a component marketplace**. This will make buying and using a software component as natural as buying a battery for a flashlight.

2.1.1.7 Simplicity

A solution that encompasses all the other requirements but is too complex will not be used. Thus our final requirement is one of simplicity. A CCA model must make sense without too much theory or special knowledge, and must be tractable for those who understand the domain, rather than the technology. It must support the construction of simple tools and techniques that assist the developer by providing a simple yet powerful paradigm. Simplicity needs to be defined in terms of the problem – how simply can the paradigm so0lve my business problems. Simplistic infrastructure and tools that make it hard to solve real problems are not viable.

2.1.2 Approach

Our approach to these requirements is to utilize the Unified Modeling Language (UML) as a basis for an architecture of recursive decomposition and assembly of parts. CCA profiles three UML diagrams and adds one optional diagram.

2.1.2.1 Class Structure (Structure)

The class structure is used to show the structure of ProcessComponents and the information which flows between them.

2.1.2.2 Statecharts (Choreography)

Statecharts are used to specify the dynamic (or temporal) contract of protocols and components, when messages should be sent or received on various ports. The Choreography specifies the intended external behavior of a component, either by specifying transitions directly on its ports or indirectly via it'a protocols.

2.1.2.3 Collaborations (Composition)

Collaborations are used to show the composition of a ProcessComponent (or community) by using a set of other ProcessComponents, configuring them and connecting them together.

2.1.2.4 CCA Notation (Structure & Composition)

CCA Also defines a notation which integrates the ProcessComponent structure and composition.

2.1.3 Concepts

At the outset it should be made clear that we are dealing with a logical concept of component - "part", something that can be incorporated in a logical composition. It is referred to in the CCA as a ProcessComponent. In some cases ProcessComponents will correspond and have a mapping to physical components and/or deployment units in a particular technology.

Since CCA, by its very nature, may be applied at many levels, it is intended that CCA be further specialized, using the same mechanisms, for specific purposes such as Business-2-Business, e-commerce, enterprise application integration (EAI), distributed objects, real-time etc.

It is specifically intended that different kinds and granularities of ProcessComponents at different levels will be joined by the recursive nature of the CCA. Thus ProcessComponents describing a worldwide B2B business process can decompose into application level ProcessComponents integrated across the enterprise which can decompose into program level ProcessComponents within a single system. However, this capability for recursive decomposition is not always required. Any ProcessComponent's part may be implemented directly in the technology of choice without requiring decomposition into other ProcessComponents.

The CCA describes how ProcessComponents at a given level of specification collaborate and how they are decomposed at the next lower level of specification. Since the specification requirements at these various levels are not exactly the same, the CCA is further specialized with profiles for each level. For example, ProcessComponents exposed on the Internet will require features of security and distribution, while more local ProcessComponents will only require a way to communicate.

The recursive decomposition of ProcessComponents utilizes two constructs in parallel: composition (using UML Collaboration) to show what ProcessComponents must be assembled and how they are put together to achieve the goal, and choreography (the UML Statechart) to show the coordination of activities to achieve a goal. The CCA integrates these concepts of "what" and "when" at each level.

Concepts from the Object Oriented Role Analysis Method (OORAM) [31] and Real-time Object Oriented Modeling (ROOM) [32] have been adapted and incorporated into CCA.

2.1.3.1 What is a Component Anyway?

There are many kinds of components – software and otherwise. A component is simply

something capable of composing into a composition – or part of an assembly. There are very different kinds of compositions and very different kinds of components. For every kind of component there must be a corresponding kind of composition for it to assemble into. Therefore any kind of component should be qualified as to the type of composition. CCA does not claim to be "the" component model, it is "a" component model with a corresponding composition model.

CCA ProcessComponents are processing components, ones that collaborate with other CCA ProcessComponents within a CCA composition. CCA ProcessComponents can be used to build other CCA ProcessComponents or to implement roles in a process – such as a vendor in a buy-sell process. The CCA concepts of component and composition are interdependent.

There are other forms of software and design components, including UML components, EJBs, COM components, CORBA components, etc. CCA ProcessComponents and composition are orthogonal to these concepts. A technology component, such as an EJB may be the implementation platform for a CCA ProcessComponent.

Some forms of components and compositions allow components to be built from other components, this is a recursive component architecture. CCA is such a recursive component architecture.

2.1.3.2 ProcessComponent Libraries

While the CCA describes the mechanisms of composition it does not provide a complete ProcessComponent library. ProcessComponent libraries may be defined and extended for various domains. A ProcessComponent library is essential for CCA to become useful without having to re-invent basic concepts.

2.1.3.3 Execution & Technology profiles

The CCA does not, in itself, specify sufficient detail to provide an executable system. However, it is a specific goal of CCA that when a CCA specification is combined with a specific infrastructure, executable primitive ProcessComponents and a *technology profile*, it will be executable.

A technology profile describes how the CCA or a specialization of CCA can be realized by a given technology set. For example, a technology profile for Java may enable Java components to be composed and execute using dynamic execution and/or code generation. A technology profile for CORBA may describe how CORBA components can be composed to create new CORBA components and systems. In RM-ODP terms, the technology profile represents the engineering and technology specifications.

Some technology profiles may require additional information in the specification to execute as desired; this is generally done using tagged values in the specification and options in the mapping. The way in which technology specific choices are combined with a CCA specification is outside of the scope of the CCA, but within the scope of the technology profile. For example, a Java mapping may provide a way to specify the signatures of methods required for Java to implement a component.

The combination of the CCA with a technology profile provides for the automated development of executable systems from high-level specifications.

For details of possible (non-normative)mappings from the CCA Profile to various engineering and technology options, see Part II of this submission.

2.1.3.4 Specification Vs. Methodology

The CCA provides a way to specify a system in terms of a hierarchical structure of Communities of ProcessComponents and Entities that, when combined with specifications prepared using technology profiles, is sufficiently complete to execute. Thus the CCA specification is the end-result of the analysis and design process. The CCA does not specify the method by which this specification is achieved. Different situations may require different methods. For example; a project involving the integration of existing legacy systems will require a different method than one involving the creation of a new real-time system – but both may share certain kinds of specification.

2.1.3.5 Notation

The CCA defines some new notations to simplify the presentation of designs for the user. These new notations are optional in that standard UML notation may be used when such is preferred or CCA specific tooling is not available. The CCA notation can be used to achieve greater simplicity and economy of expression.

2.1.4 Conceptual Framework

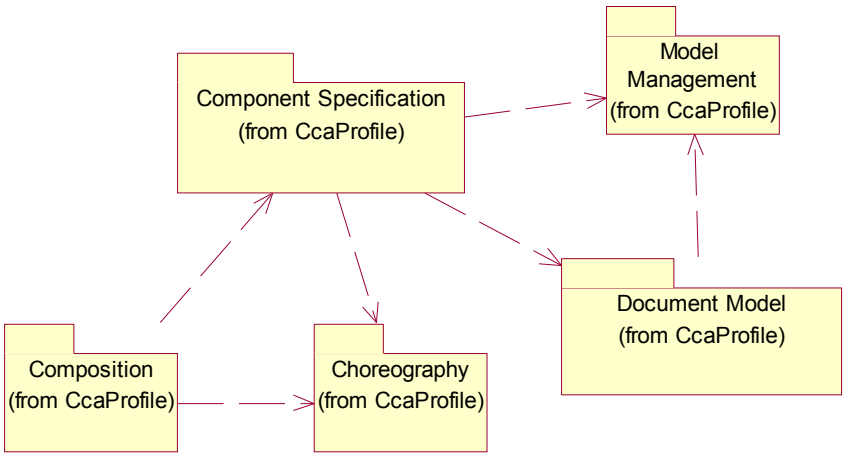

Figure 6: Structure and dependencies of the CCA Metamodel

2.1.4.1 ProcessComponent Specification

In keeping with the concept of encapsulation, the external "contract" of a CCA component is separate from how that component is realized. The contract specifies the "outside" of the component. Inside of a component is its realization – how it satisfies its contract. The outside of the component is the **component specification**. A component with only a specification is *abstract*, it is just the "outside" with no "inside".

2.1.4.2 Protocols and Choreography

Part of a component's specification is the set of **protocols** it implements. A protocol specifies what messages the component sends and receives when it collaborates with another component and the **choreography** of those messages – when they can be sent

and received. Each protocol the component supports is provided via a "**port**", the connection point between components.

Protocols, ports and choreography comprise the contract on the outside of the component. Protocols are also used for large-grain interactions, such as for B2B components.

The protocol specifies the conversation between two components (via their ports). Each component that is using that protocol must use it from the perspective of the "initiating role" or the "responding role". Each of these components will use every port in the protocol, but in complementary directions.

For example, a protocol "X" has a flow port "A" that initiates a message and a flow port "B" that responds to a message. Component "Y" which responds to protocol "X" will also receive "A" and initiate "B". But, Component "Z" which initiates protocol "X" will also initiate message "A" and respond to message "B" – thus initiating a protocol will "invert" the directions of all ports in the protocol.

2.1.4.3 Primitive and Composed Components

Components may be abstract (having only an outside) or concrete (having an inside and outside). Frequently a concrete component inherits its external contract from an abstract component – implementing that component.

There may be any number of implementations for an **ProcessComponent** and various ways to "bind" the correct implementation when a component is used.

The two basic kinds of concrete components are:

- **primitive components** – those that are built with programming languages or by wrapping legacy systems.
- **Composed Components** – Components that are built from other components; these use other components to implement the new components functionality. Composed components are defined using a **composition**.

2.1.4.4 Composition

Compositions define how components are *used*. Inside of a composition components are used, configured and connected. This connected set of component usages implements the behavior of the composition in terms of these other components – which may be primitive, composed or abstract components.

Compositions are used to build composed components out of other components and to describe community processes – how a set of large grain components works together for some purpose. Components used in a community process represent the roles of that process.

Central to compositions are the **connections** between components, values for **configuration properties** and the ability to **bind** concrete components to a component usage.

2.1.4.5 Document & Information Model

The information that flows between components is described in a **Document Model**, the structure of information exchanged. The document model also forms the basis for information entities and a generic **information model**. The information model is acted on by CCA ProcessComponents (see the Entities profile, Section 3, below).

2.1.4.6 Model Management

To help organize the elements of a CCA model a "**package**" structure is used exactly as it is used in UML. Packages provide a hierarchical name space in which to define components and component artifacts. Model elements that are specific to a process, protocol or component may also be nested within these, since they also act as packages.

2.2 CCA Metamodel

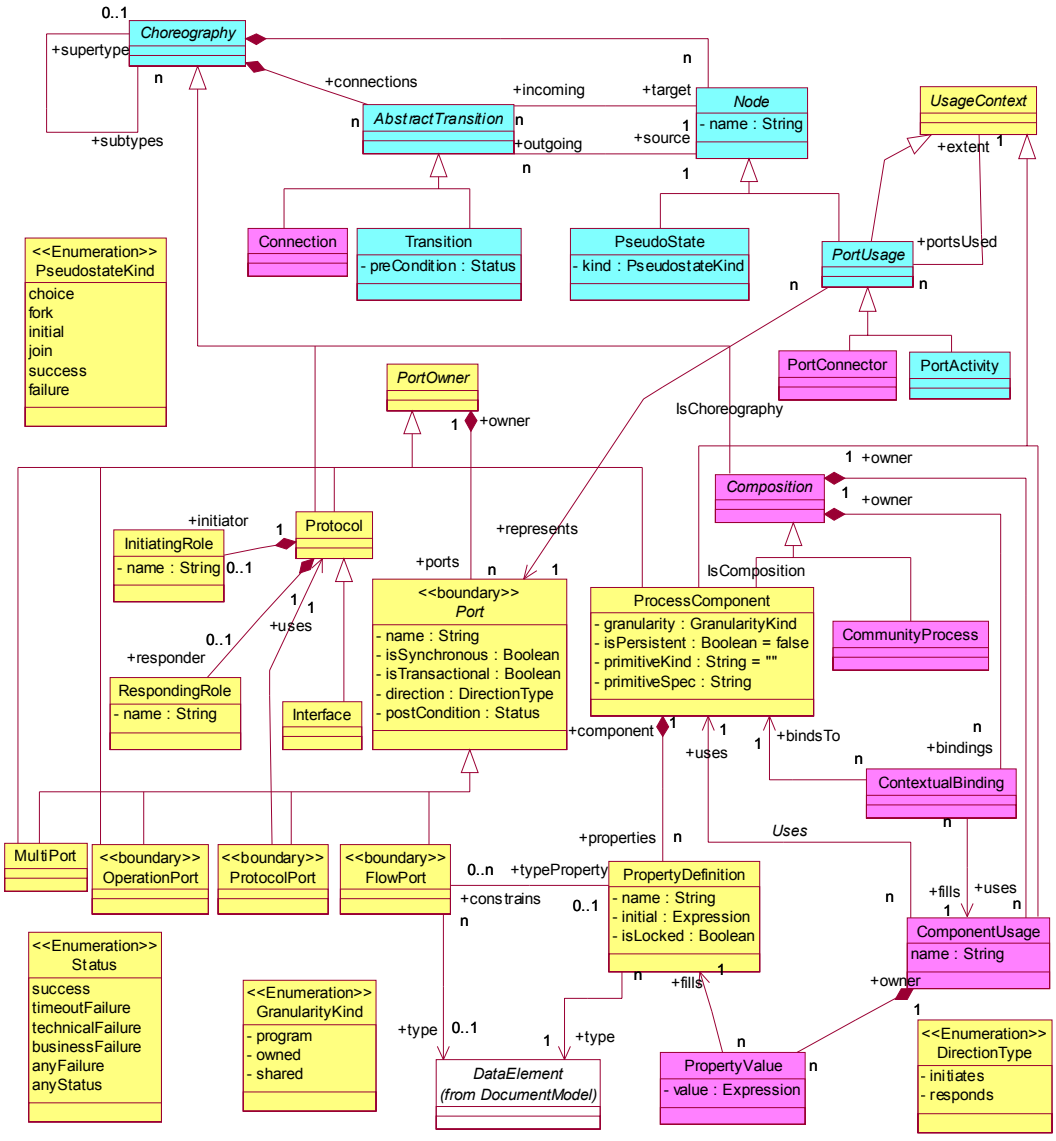

Figure 7: CCA Major elements

Figure 7 above is a combined model of the major elements of the CCA component specification defined below.

2.2.1 Structural Specification

The structural specification represents the physical structure of the component contract, defining the component and its ports.

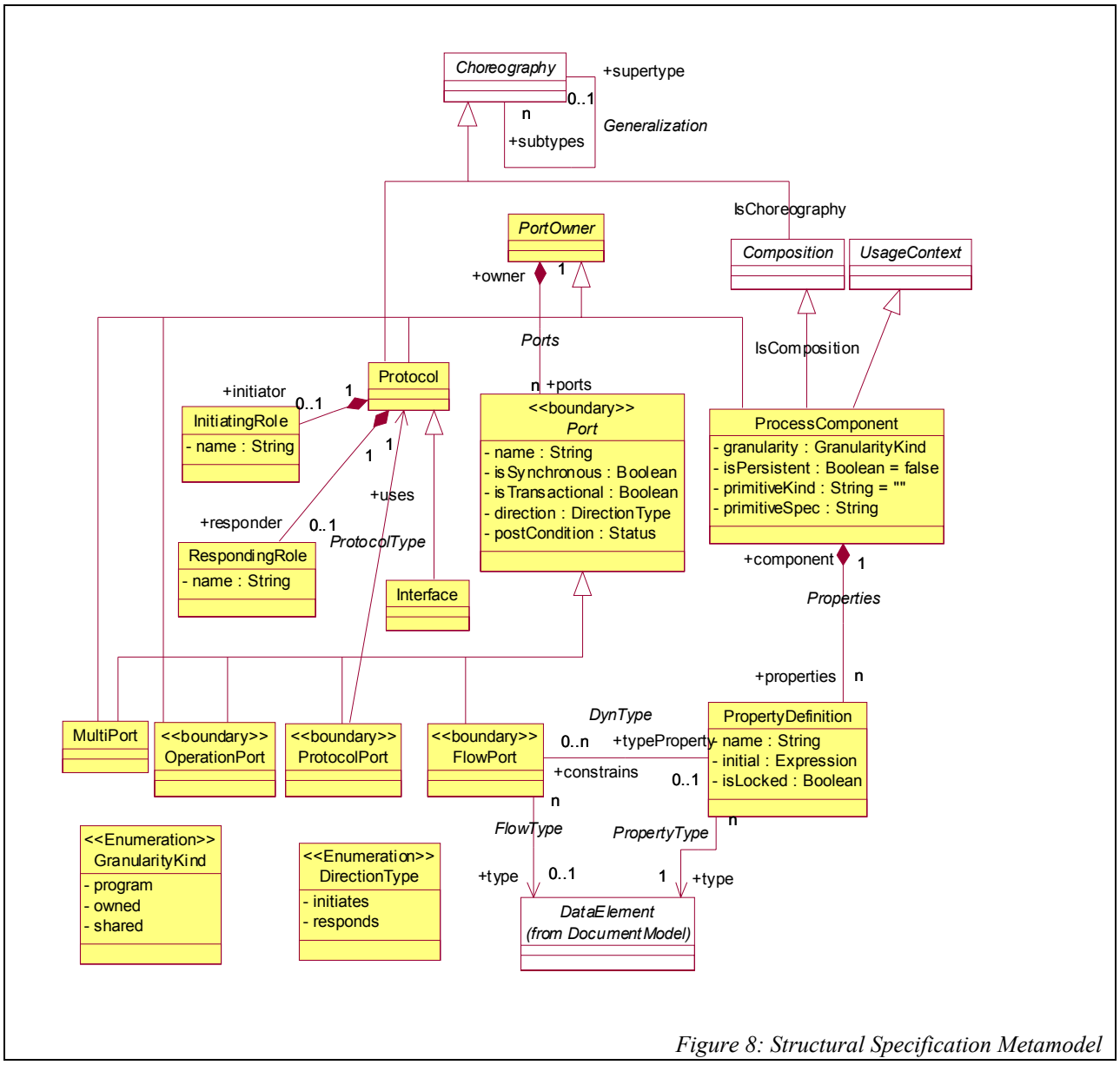

Figure 8: Structural Specification Metamodel

A **ProcessComponent** represents the contract for a component that performs actions – it "does something". A ProcessComponent may define a set of **Port**s for interaction with other ProcessComponents. The ProcessComponent defines the external contract of the component in terms of ports and a **Choreography** of port activities (sending or receiving messages or initiating sub-protocols). At a high level of abstraction a ProcessComponent can represent a business partner, other ProcessComponents represent business activities or finer-grain capabilities.

The contract of the ProcessComponent is realized via **ports**. A port defines a point of interaction between ProcessComponents. The simpler form of port is the **FlowPort**, which may produce or consume a single **data type**. More complex interactions between components use a **ProtocolPort**, which refers to a **Protocol,** a complete "conversation" between components. Protocols may also use other protocols as sub-protocols. Protocols, like ProcessComponents, are defined in terms of the set of ports

they realize and the choreography of interactions across those ports. A protocol may optionally define names for the initiating and responding roles.

ProcessComponents may have **Property Definitions**. A property definition defines a configuration parameter of the component, which can be set, when the component is used.

The behavior of a ProcessComponent may be further specified by its **composition**, the composition shows how other components are used to define and implement the composite component. The specification of the ProcessComponent and protocol may include **Choreography** to sequence the actions of multiple ports and their associated actions. The actions of each port may be **Choreographed**. Composition and Choreography are defined in their own sections.

A ProcessComponent may have a **supertype** (derived from Choreography). One common use of supertype is to place abstract ProcessComponents within compositions and then produce separate realizations of those components as subtype composite or primitive components, which can then be substituted for the abstract components when the composition is used, or even at runtime.

An **Interface** represents a standard object interface. It may contain **OperationPorts**, representing call-return semantics, and FlowPorts – representing one-way operations.

A **MultiPort** is a grouping of ports whose actions are tied together. Information must be available on all sub-ports of the MultiPort for any action to occur within an attached component.

An **OperationPort** defines a port which realizes a typical request/response operation and allows ProcessComponents to represent both document oriented (FlowPort) and method oriented (OperationPort) subsystems.

2.2.1.1 ProcessComponent

Semantics

A ProcessComponent represents an active processing unit – it does something. A ProcessComponent may realize a set of Ports for interaction with other ProcessComponents and it may be configured with properties.

Each ProcessComponent defines a set of ports for interaction with other ProcessComponents and has a set of properties that are used to configure the ProcessComponent when it is used.

The order in which actions of the Process Component's ports do something may be specified using Choreography. The choreography of a ProcessComponent specifies the external temporal contact of the ProcessComponent (when it will do what) based on the actions of its ports and the ports in protocols of its ports.

UML base element(s) in the Profile and Stereotype

Classifier Stereotyped as <<ProcessComponent>>

Fully Scoped name

ECA::CCA::ProcessComponent

Owned by

Package

Extends

Composition (indicating that the ProcessComponent may be composed of other ProcessComponents and that its ports may be choreographed.)

Package (Indicating that a ProcessComponent may own the specification of other elements)

UsageContext (Indicating that the ProcessComponent may be the context for PortUsages representing the activities of its ports.).

Properties

Granularity

A GranularityKind which defines the scope in which the component operates. The values may be:

- **Program** – the component is local to a program instance (default)
- **Owned** – the component is visible outside of the scope of a particular program but dedicated to a particular task or session which controls its life cycle.
- **Shared** – the component is generally visible to external entities via some kind of distributed infrastructure.

Specializations of CCA may define additional granularity values.

UML Representation

Tagged value

isPersistent

Indicates that the component stores session specific state across interactions. The mechanisms for management of sessions are defined outside of the scope of CCA.

UML Representation

Tagged value

primitiveKind

Components implementation includes additional implementation semantics defined elsewhere, perhaps in an action language or programming language. If the component has an implementation specification primitiveKind specifies the implementation specific type, normally the name of a programming language. If primitive kind is blank, the composition is the full specification of the components implantation – the component is not primitive.

UML Representation

Tagged value

primitiveSpec

If primitiveKind has a value, primitiveSpec identifies the location of the implementation. The syntax of primitiveKind is implementation specific.

UML Representation

Tagged value

Related elements

Ports (via "PortOwner")

"Ports" is the set of Ports on the ProcessComponent. Each port provides a connection point for interaction with other components or services and realizes a specific protocol. The protocol may be simple and use a "FlowPort" or the protocol may be complex and use a "ProtocolPort" or an "OperationPort". If allowed by its protocol, a port may send and receive information.

UML Representation

Required Aggregation Association from Port (Ports)

Supertype (zero or one) , Subtypes (any number)

A ProcessComponent may inherit specification elements (ports, properties & states (from Choreography) from a supertype. That supertype must also be a ProcessComponent. A subtype component is bound by the contract of its supertypes but it may add elements, override property values and restrict referenced types.

A component may be substituted by a subtype of that component.

UML Representation

Generalization

Properties (Any number)

To make a component capable of being reused in a variety of conditions it is necessary to be able to define and set properties of that component. Properties represents the list of properties defined for this component.

UML Representation

Classifier.feature referencing an attribute.

Constraints

A process component may only inherit from another process component.

2.2.1.2 Port

Semantics

A port realizes a simple or complex conversation for a ProcessComponent or protocol. All interactions with a ProcessComponent are done via one of its ports.

When a component is instantiated, each of its ports is instantiated as well, providing a well-defined connection point for other components.

Each port is connected with collaborative components that speak the same protocol. Multi-party conversions are defined by components using multiple ports, one for each kind of party.

Business Example: Flight reservation Port

UML base element(s) in the Profile and Stereotype

Class (abstract)

Fully Scoped name

ECA::CCA::Port

Owned by

ProcessComponent or Protocol via PortOwner

Extends

None

Properties

isTransactional

Indicates that interactions with the component are transactional & atomic (in most implementations this will require that a transaction be started on receipt of a message). Non-transactional components either maintain no state or must execute within a transactional component. The mechanisms for management of transactions are defined outside of the scope of CCA.

UML Representation

Tagged Value

isSynchronous

A port may interact synchronously or asynchronously. A port that is marked as synchronous is required to interact using synchronous messages and return values.

UML Representation

Tagged Value

name

The name of the port. The name will, by default, be the same as the name of the protocol role or document type it realises.

UML Representation

ModelElement::name

Direction

Indicates that the port will either initiate or respond to the related type. An initiating port will send the first message. Note that by using ProtocolPorts a port may be the initiator of some protocols and the responder to others. The values of DirectionKind may be:

Initiates – this port will initiate the conversation by sending the first message.

Responds – this port will respond to the initial message and (potentially) continue the conversation.

UML Representation

Tagged Value and stereotype of "Owner" relation.

PostCondition

The status of the conversation indicated by the use of this port. This status may be queried in the postCondition of a transition.

UML Representation

Tagged Value

Related elements

"Owner" ProcessComponent or Protocol (Exactly One via PortOwner)

A Port specifies the realization of protocol by a ProcessComponent. This relation specifies the ProcessComponent that realizes the protocol.

UML Representation

Required aggregate association (Ports). This association will have a stereotype of "initiates" or "responds" to indicate "direction".

Constraints

None

2.2.1.3 FlowPort

Semantics

A Flow Port is a port which defines a data flow in or out of the port on behalf of the owning component or protocol.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <<FlowPort>>

Fully Scoped name

ECA::CCA::FlowPort

Owned by

PortOwner

Extends

Port

Properties

None

Related elements

type

The type of data element that may flow into our out of the port.

UML Representation

Required relation

TypeProperty

The type of information sent or received by this port as determined by a configurable property. The expression must return a valid type name. This is used to build generic components that may have the type of their ports configured. If *type* and *typeProperty* are both set then the property expression must return the name of a subtype of *type*.

UML Representation

Tagged value containing the name of the property attribute.

Constraints

None

2.2.1.4 ProtocolPort

Semantics

A protocol port is a port which defines the use of a protocol A protocol port is used for potentially complex two-way interactions between components, such as is common in B2B protocols. Since a protocol has two "roles" (the initiator and responder), the direction is used to determine which role the protocol port is taking on.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <<ProtocolPort>>

Fully Scoped name

ECA::CCA::ProtocolPort

Owned by

PortOwner

Extends

Port

Properties

None

Related elements

uses

The protocol to use, which becomes the specification of this port's behavior.

UML Representation

Generalization – the ProtocolPort inherits the Protocol.

Constraints

None

2.2.1.5 OperationPort

Semantics

An operation port represents the typical call/return pattern of an operation. The OperationPort is a PortOwner which is constrained to contain only flow ports, exactly one of which must have its direction set to "initiates". The other "responds" ports will be the return values of the operation.

UML base element(s) in the Profile and Stereotype

Operation (no stereotype)

Note1: The type of the "initiates" flow port will be the signature of the operation. Each attribute of the type will be one parameter of the operation.

Note2: Owned flow ports of postCondition==Success and direction=="responds" will be a return value for the operation. All other flow ports where direction=="responds" will correspond to an exception.

Fully Scoped name

ECA::CCA::OperationPort

Owned by

PortOwner (Protocol or ProcessComponent)

Extends

Port and PortOwner

Properties

None

Related elements

Ports (Via PortOwner)

The flow ports representing the call and returns.

UML Representation

Initiates ports – signature of the operation

Responds ports – return values of the operation.

Constraints

As a PortOwner, the OperationPort:

- May only contain FlowPorts
- Must contain exactly one flow port with direction set to "responds" (the call)

2.2.1.6 MultiPort

Semantics

A MultiPort combines a set of ports which are behaviourally related. Each port owned by the MultiPort will “buffer” information sent to that port until all the ports within the MultiPort have received data, at this time all the ports will send their data.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <<MultiPort>>

Fully Scoped name

ECA::CCA::MultiPort

Owned by

PortOwner

Extends

Port & PortOwner

Properties

None

Related elements

Ports (Via PortOwner)

The flow ports owned by the MultiPort.

UML Representation

Required aggregation association

Constraints

Owned ports will not forward data until all sub-ports have received data.

2.2.1.7 Protocol

Semantics

A **protocol** defines a type of conversation between two parties, the initiator and responder. One protocol role is the initiator of the conversation and the other the responder. However, after the conversation has been initiated, individual messages and sub-protocols may by initiated by either party. The ports of a protocol are specified with respect to the responder.

Within the protocol are sub-ports . Each port contained by a protocol defines a sub-action of that protocol until, ultimately, everything is defined in terms of FlowPorts.

A Protocol is also a choreography, indicating that activities of its ports (and, potentially their sub-ports) may be sequenced using an activity graph.

A protocol must be used by two ProtocolPorts to become active.

The protocol specifies the conversation between two ProcessComponents (via their ports). Each component that is using that protocol must use it from the perspective of the "initiating role" or the "responding role". Each of these components will use every port in the protocol, but in complementary directions.

For example, a protocol "X" has a flow port "A" that initiates a message and a flow port "B" that responds to a message. Component "Y" which responds to protocol "X" will also receive "A" and initiate "B". But, Component "Z" which initiates protocol "X" will initiate message "A" and respond to message "B" – thus initiating a protocol will "invert" the directions of all ports in the protocol.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <<Protocol>>

Fully Scoped name

ECA::CCA::Protocol

Owned by

Package

Extends

Choreography – Indicating that the contract of the protocol includes a sequencing of the port activities.

Package – Indicating that the protocol may contain the specification of other model elements (Most probably other protocols or documents).

Properties

None

Related elements

Ports (Via PortOwner)

The ports which define the sub-actions of the protocol. For example, a "callReturn" protocol may have a "call" FlowPort and a "return" FlowPort.

UML Representation

Required aggregate association

Initiator

The role which sends the first message in the protocol. Note that this is optional, in which case the initiating role name will be "Initiator".

UML Representation

Required relation

Responder

The role which receives the first message in the protocol. Note that this is optional, in which case the responding role name will be "Responder".

UML Representation

Required relation

Constraints

None

2.2.1.8 Interface

Semantics

An interface is a protocol constrained to match the capabilities of the typical object interface. It is constrained to only contain OperationPorts and FlowPorts and all of its ports must respond to the interaction (making interfaces one-way).

Each OperationPort or FlowPort in the Interface will map to a method. A ProtocolPort which initiates the Interface will call the interface. A ProtocolPort which Responds will implement the interface.

UML base element(s) in the Profile and Stereotype

Classifier (Usually Interface, but any classifier will do)

Fully Scoped name

ECA::CCA::Interface

Owned by

Package

Extends

Protocol

Properties

None

Related elements

Ports (Via Protocol & PortOwner)

The ports which define the sub-actions of the protocol. For example, a “callReturn” protocol may have a “call” flowport and a “return” port.

Initiator (Via Protocol)

The role which calls the interface. Note that this is optional, in which case the initiating role name will be “Initiator”. roles.

Responder (Via Protocol)

The role which implements the interface. Note that this is optional, in which case the responding role name will be “Responder”.

Constraints

- The Ports related by the “Ports” association must;
  - be of type OperationPort or FlowPort.
  - have direction == ”responds”.

2.2.1.9 InitiatingRole

Semantics

The role of the protocol which will send the first message.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <InitiatingRole>

Fully Scoped name

ECA::CCA::InitiatingRole

Owned by

Protocol

Extends

None

Properties

name

Role name

UML Representation

ModelElement::name

Related elements

Protocol

The protocol for which the role is being defined.

UML Representation

Required relation

Constraints

None

2.2.1.10 RespondingRole

Semantics

The role in the protocol which will receive the first message.

UML base element(s) in the Profile and Stereotype

Class stereotyped as <RespondingRole>

Fully Scoped name

ECA::CCA::RespondingRole

Owned by

Protocol

Extends

None

Properties

Name

UML Representation

ModelElement::name

Related elements

Protocol

The protocol for which the role is being defined.

UML Representation

Required relation

Constraints

None

2.2.1.11 PropertyDefinition

Semantics

To allow for greater flexibility and reuse, ProcessComponents may have properties which may be set when the ProcessComponent is used. A **PropertyDefinition** defines that such a property exists, its name and type.

UML base element(s) in the Profile and Stereotype

Attribute (No stereotype)

Fully Scoped name

ECA::CCA::PropertyDefinition

Owned by

ProcessComponent

Extends

None

Properties

name

Name of the property being modelled

UML Representation

ModelElement:name

initial

An expression indicating the initial & default value.

UML Representation

Attribute::initialValue

isLocked

The property may not be changed.

UML Representation

StructuralFeature::changeability

Related elements

component

The owning component

UML Representation

Classifier.feature referencing an attribute.

type

The type of the property

UML Representation

StructuralFeature::type

Constraints

If the “constrains” relation contains any links;

- The PropertyValue must contain the fully qualified name of a DataElement.

PortOwner

Semantics

An abstract meta-class used to group the meta-classes that may own ports: Process component, Protocol, OperationPort and MultiPort.

UML base element(s) in the Profile and Stereotype

None (Abstract)

Fully Scoped name

ECA::CCA::PortOwner

Owned by

None

Extends

None

Related elements

ports

The owned ports

UML Representation

Required relation

Constraints

None

2.2.2 Choreography

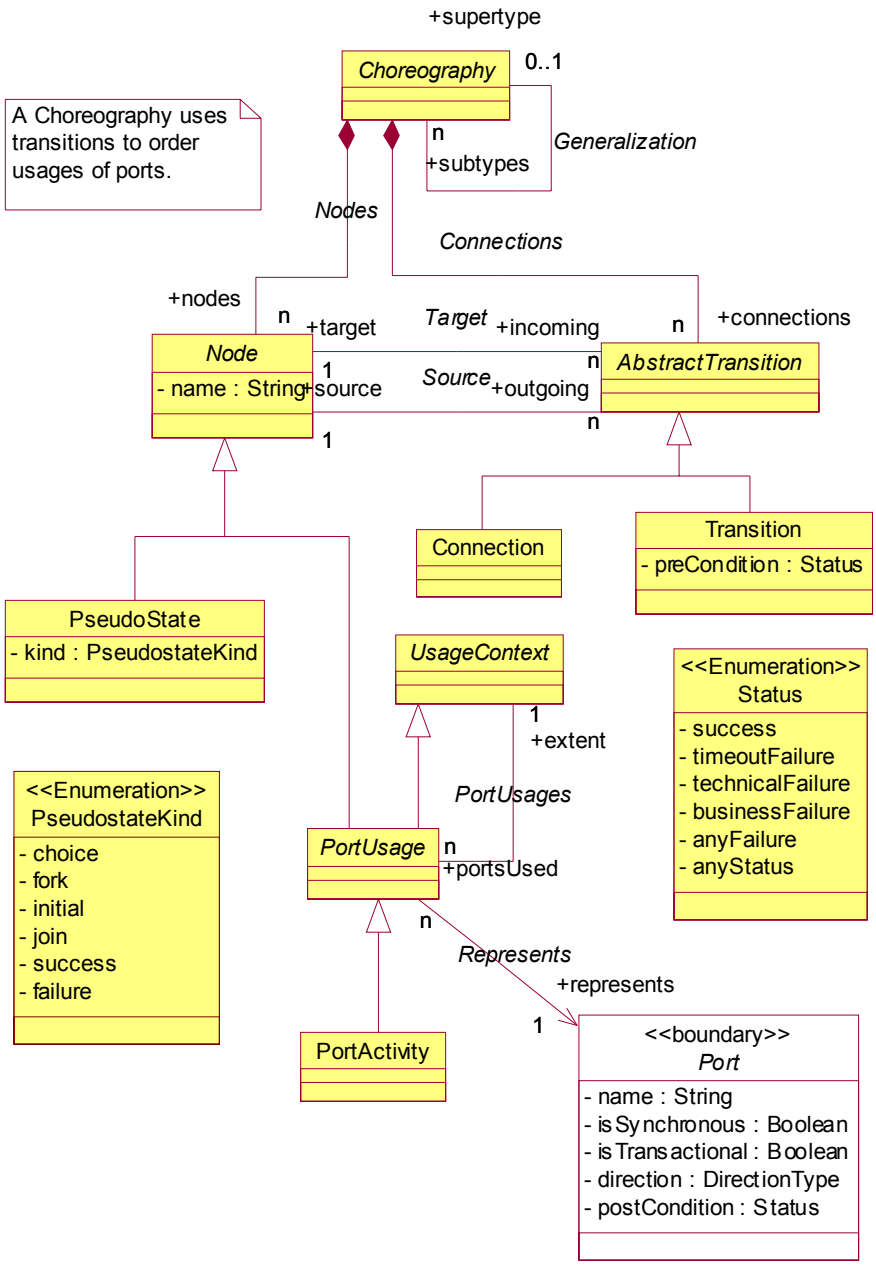

Figure 9: Choreography Metamodel

A Choreography specifies how messages will flow between PortUsages. The choreography may be externally oriented, specifying the contract a component will have with other components or, it may be internally oriented, specifying the flow of messages within a composition. External chirographies are shown as an activity

graph while internal choreography is shown as part of a collaboration. An external choreography may be defined for a protocol or a ProcessComponent.

A **Choreography** uses **Connections** and **transitions** to order port messages as a state machine. Each "node" in the choreography must refer to a state or a port usage.

Choreography is an abstract capability that is inherited by ProcessComponents and protocols.

Initial, interim and terminating states are known as a "**PseudoState**" as defined in UML. CCA adds the pseudo states for success and failure end-states.

Ordering is controlled by **connections** between nodes (state and port usage being a kind of node). Transitions specify flow of control that will occur if the conditions (Precondition) are met. Transitions between port activities specify what should happen (contractually), while Connections between PortConnections specify what will happen at runtime.

2.2.2.1 Choreography

Semantics

An abstract class inherited by protocol and ProcessComponent which owns nodes and AbstractTransitions. A choreography specifies the ordering of port activities.

UML base element(s) in the Profile and Stereotype

Choreography - State Machine stereotyped as <<choreography>>: (context references classifier)

Fully Scoped name

ECA::CCA::Choreography

Owned by

None

Extends

None

Properties

None

Related elements

Nodes

The states and port usages to be choreographed.

UML Representation

PseudoState - StateMachine.top

PortActivity ::SubmachineState

AbstractTransitions

The connections and transitions between nodes.

UML Representation

Transition: StateMachine:transition

Connection: Collaboration::AssociationRole

Supertype (zero or one) , Subtypes (any number)

A ProcessComponent, protocol or CommunityProcess may inherit specification elements (ports, properties & states (from Choreography) from a supertype. That supertype must also be a ProcessComponent. A subtype component is bound by the contract of its supertypes but it may add elements, override property values and restrict referenced types.

A component may be substituted by a subtype.

Constraints: The subtype-supertype relation may only exist between elements of the same meta-type. A ProcessComponent may only inherit from another ProcessComponent. A Protocol may only inherit from another Protocol and a CommunityProcess may only inherit from another CommunityProcess.

UML Representation

Generalization of classifier related by context.

2.2.2.2 Node

Semantics

Node is an abstract element that specifies something that can be the source and/or target of a connection or transition and thus ordered within the choreographed process. The nodes that do “real work” are PortUsages.

UML base element(s) in the Profile and Stereotype

None (abstract)

Fully Scoped name

ECA::CCA::Node

Owned by

Choreography

Extends

None

Properties

name

UML Representation

ModelElement:name

Related elements

Choreography

The owning protocol or ProcessComponent.

UML Representation

See Choreogrphy

Incoming

Transitions that cause this node to become active.

UML Representation

Transition: State:incoming

Connection: AssociationEndRole

outgoing

Nodes that may become active after this node completes.

UML Representation

State: outgoing

Connection: AssociationEndRole

Constraints

None

2.2.2.3 AbstractTransition

Semantics

The flow of data and/or control between two nodes.

UML base element(s) in the Profile and Stereotype

None - abstract

Fully Scoped name

ECA::CCA::AbstractTransition

Owned by

Choreography

Extends

None

Properties

None

Related elements

Choreography

The owning choreography.

UML Representation

See Choreography

Source

The node which is transferring control and/or data.

UML Representation

Connection: AssociationEndRole

Transition: Transition:source

Target

The node to which data and/or control will be transferred.

UML Representation

Connection: AssociationEndRole

Transition: Transition:target

Constraints

The source and target nodes associated with the AbstractTransition must be owned by the same choreography as the AbstractTransition.

2.2.2.4 Transition

Semantics

The contractual specification that the related nodes will activate based on the ordering imposed by the set of transitions between nodes. Transitions, which declare a contract may be differentiated from Connections which realise a contract.

UML base element(s) in the Profile and Stereotype

Transition (No Stereotype)

Fully Scoped name

ECA::CCA::Transition

Owned by

Choreography

Extends

AbstractTransition

Properties

preCondition

A constraint on the transition such that it may only fire if the prior PortUsage terminated with the referenced condition.

UML Representation

Transition:guard

Related elements

Choreography (Via AbstractTransition)

The owning choreography.

UML Representation

See Choreography

Source

The node which is transferring control and/or data.

UML Representation

Transition: Transition:source

Target

The node to which data and/or control will be transferred.

UML Representation

Transition: Transition:target

Constraints

A transition may not connect PortConnectors.

2.2.2.5 PortUsage

Semantics

The usage of a port as part of a choreography.

UML base element(s) in the Profile and Stereotype

None (Abstract)

Fully Scoped name

ECA::CCA::PortUsage

Owned by

Choreography

Extends

Node & Usage Context

Properties

None

Related elements

extent

The component, component usage or PortUsage to which the PortUsage is attached.

If the extent is a ComponentUsage the PortUsage must be a PortConnector for a port of the underlying ProcessComponent. This allows Connections between components being used within a composition.

If the extent is a PortUsage the PortUsage must represent a ProtocolPort which owns the represented usage. This allows the choreography of nested ports.

If the extent is a ProcessComponent the usage represents a port on the ProcessComponent and that ProcessComponent must be the composition owning both the port and the port usage. This allows Connections and transitions to be connected to the external ports of a component.

UML Representation

State machine: Owner of state machine

Collaboration: Association Role

Represents

The port which the PortUsage uses.

UML Representation

State machine: tagged value

Collaboration: ClassifierRole::base

Constraints

None

2.2.2.6 UsageContext

Semantics

When a port is used within a choreography it must be used within some context. UsageContext represents an abstract supertype of all elements that may be the context of a port. These are;

- ProcessComponent – as the owner of port activities and port connectors.
- ComponentUsage – as the owner of port connectors, representing the use of each of the component's ports.
- PortUsages – representing ports nested via protocols.

UML base element(s) in the Profile and Stereotype

None (abstract)

Fully Scoped name

ECA::CCA::UsageContext

Owned by

None

Extends

None

Properties

None

Related elements

PortsUsed

Provides context for port usage

UML Representation

State machine: owned states

Collaboration: AssociationRole

Constraints

None

2.2.2.7 PortActivity

Semantics

Port activity is state, part of the "contract" of a ProcessComponent or protocol, specifying the activation of a port such the ordering of port activities can be choreographed with transitions. A PortActivity (used with transitions) defines the contract of the component while a PortConnector (used with Connections) specifies the realisation of a component's actions in terms of other components.

UML base element(s) in the Profile and Stereotype

CompositeState Stereotyped as <<PortActivity>>

Fully Scoped name

ECA::CCA::PortActivity

Owned by

Protocol or ProcessComponent via Choreography

Extends

PortUsage

Properties

None

Related elements

None

Constraints

Port Activities may only be connected using transitions.

2.2.2.8 PseudoState

Semantics

PseudoState specifies starting, ending or intermediate states in the choreography of the contract of a protocol or ProcessComponent.

UML base element(s) in the Profile and Stereotype

Depending on value of kind:

- **Success** – FinalState Stereotyped as <<success>>
- **Failure** – FinalState Stereotyped as <<failure>>
- **All Others** - PseudoState (no stereotype) with kind set to same value.

Fully Scoped name

ECA::CCA::PseudoState

Owned by

Choreography

Extends

Node

Properties

Kind ; PseudostateKind

choice Splits an incoming transition into several disjoint outgoing transition. Each outgoing transition has a guard condition that is evaluated after prior actions on the incoming path have been completed. At least one outgoing transition must be enabled or the model is ill-formed.

fork Splits an incoming transition into several concurrent outgoing transitions. All the transitions fire together.

initial The default target of a transition to the enclosing composite state.

join Merges transitions from concurrent regions into a single outgoing transition. Join PseudoState will proceed after all its incoming Transition have triggered.

success The end-state indicating that the choreography ended in success.

failure The end-state indicating that the choreography ended in failure.

Related elements

None

Constraints

PseudoStates may only be connected using transitions.

2.2.3 Composition

Composition is an abstract capability that is used for ProcessComponents and for community processes. Compositions shows how a set of components can be used to define and perhaps to implement a process.

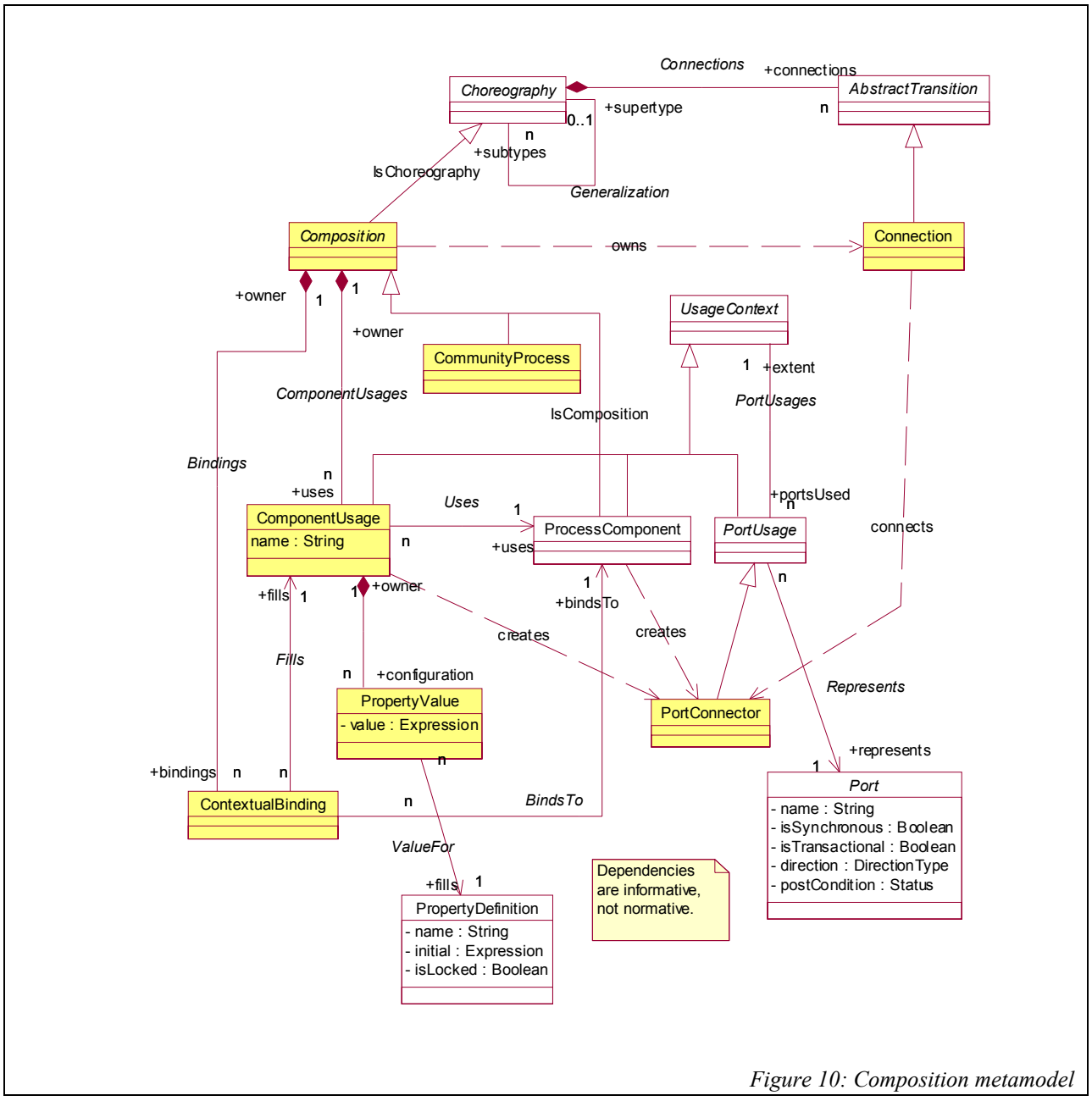

Figure 10: Composition metamodel

A **composition** contains **ComponentUsage**s to show how other ProcessComponents may be used to define the composite. Note that the same ProcessComponent may be used multiple times for different purposes. Each time a ProcessComponent is used, each of its ports will also be used with a "**PortConnector**". A port connector shows the connection point for each use of that component within the composition, including the ports on the component being defined.

Attached to a ProcessComponent usage are **PropertyValues**, configuring the ProcessComponent with properties that have been defined in property definitions.

A composition also contains a set of "**Connections**". A connectionjoins compatible ports on ProcessComponents together to define a flow of data. The other side will

receive anything sent out of one side. So a Connection is a form of logical event registration (one-way registration for a flow port or Operation port, two-way registration for a ProtocolPort).

A **Contextual Binding** allows realized ProcessComponents to be substituted for abstract ProcessComponents when a composition is used.

Compositions may be ProcessComponents or CommunityProcesses. **CommunityProcess** define a top-level process in terms of the roles played by process components representing actors in the process.

2.2.3.1 Composition

Semantics

Composition is an abstract class for CommunityProcesses or ProcessComponents. Compositions describe how instances of ProcessComponents (called ComponentUsages) are configured (with PropertyValues and ContextualBindings) and connected (with Connections) to implement the composed ProcessComponent or CommunityProcess.

UML base element(s) in the Profile and Stereotype

Collaboration (with represented classifier being the ProcessComponent or CommnityProcess being defined) – stereotyped as <<Composition>>

Fully Scoped name

ECA::CCA::Compsition

Owned by

None

Extends

Choreography

Properties

None

Related elements

bindings

ContextualBindings defined within the context of the composition.

UML Representation

ModelElement::clientDependency

uses

ComponentUsages defined within the context of the composition.

UML Representation

Collaboration:: (Owned ClassifierRoles)

Connection (via choreography and AbstractTransition)

The flow of data and control between port connectors.

UML Representation

Collaboration:: ownedElement (Owned AssociationRoles)

PortConnector (via Choreography and nodes)

The port instances to be connected by Connections.

UML Representation

Collaboration:: (Owned ClassifierRoles)

Constraints

None

2.2.3.2 ComponentUsage

Semantics

A composition *uses* other ProcessComponents to define the process of the composition (a community process or ProcessComponent), "ComponentUsage" represents such a use of a component. The "uses" relation references the kind of component being used. Component Usage is part of the "inside" of a composed component.

The composition can be thought of as a template of ProcessComponent instances. Each component instance will have a "ComponentUsage" to say what kind of ProcessComponent it is, what its property values are and how it is connected to other ProcessComponents. A ComponentUsage will cause a ProcessComponent instance to be created at runtime (this instantiation may be real or virtual).

Each use of a ProcessComponent will carry with it a set of "portConnectors" which will be the connection points to other ProcessComponents.

UML base element(s) in the Profile and Stereotype

ClassifierRole Stereotyped as “ComponentUsage”

Fully Scoped name

ECA::CCA::ComponentUsage

Owned by

Composition

Extends

UsageContext

Properties

Name

The name of the activity for which the component is being used.

UML Representation

ModelElement::name

Related elements

owner

The owning composition

UML Representation

ClassifierRole::(owning collaboration)

Uses

The type of ProcessComponent to use.

UML Representation

ClassifierRole::base

PortsUsed (Via UsageContext)

PortConnectors for each port on the used component.

UML Representation

AssociationRole

Constraints

None

2.2.3.3 PortConnector

Semantics

The PortConnector provides a "connection point" for ComponentUsages within a composition and exposes the defined ports within the composition. The connections between PortConnectors are made with Connections.

PortConnections are "implied" by other model elements and will normally be created by design tools. PortConnections should be created as follows:

- For each ComponentUsage there will be exactly one PortUsage for each port defined for the ProcessComponent being used.
- For each port on the ProcessComponent being defined there will be exactly one PortUsage to support Connections to and from "outside" ports.
- For each port within a protocol, OperationPort or MultiPort created for one of the above two reasons, a PortConnector may be created for each contained port. This allows Connections to be connected to finer grain elements, such as Connections within a protocol.

In summary, the "ProcessComponent" / "Port" pattern which defines the components external interface is essentially replicated in the "ComponentUsage" / "portConnector" part of the composition. Each time a component is used, each of its ports is used as well. Sub-ports of protocols also become PortConnectors.

UML base element(s) in the Profile and Stereotype

ClassifierRole stereotyped as PortConnector

Fully Scoped name

ECA::CCA::PortConnector

Owned by

Composition

Extends

PortUsage

Properties

None

Related elements

Represents (via PortUsage)

The port of which this is a port.

Contexts (via PortUsage)

The associated owner of the port.

Incoming and Outgoing Connections (Via PortUsage and Node)

The Connections.

Constraints

PortConnectors are intended to be connected with Connections, Transitions may not be connected to a PortConnector

2.2.3.4 Connection

Semantics

A Connection connects two PortConnectors within a composition. Each port can produce and/or consume message events. The connection logically registers each port connector as a listener to the other, effectively making them collaborators.

A component only declares that given ports will produce or consume given messages, it doesn't not know "who" will be on the other side. The composition shows how a ProcessComponent will be used within a context and thus how it will be connected to other components *within that context*. A Connection connects exactly two PortConnectors.

Connections may be distinguished from transitions in that Connections specify what events will flow between ProcessComponents while transitions specify the contract of port ordering.

UML base element(s) in the Profile and Stereotype

AssociationRole optionally stereotyped as <<Connection>>

Note: A Connection to a port contained by an interface will be represented by an operation, not a classifier. In this case the association role is directed to the ProtocolPort realising the interface and a message attached with a call action referencing the operation in question.

Fully Scoped name

ECA::CCA::Connection

Owned by

Composition

Extends

AbstractTransition

Properties

None

Related elements

Source and Target PortConnectors (Via PortUsage, Node & AbstractTransition)

The PortConnectors between which the Connection is being defined.

Constraints

- The source and target nodes of a Connection must be PortConnectors.
- The source and target nodes must be port connectors owned by the same composition as the Connection.

2.2.3.5 PropertyValue

Semantics

To be useful in a variety of conditions, a ProcessComponent may have configuration properties –which are defined by a PropertyDefinition. When the component is used in a ComponentUsage those properties values may be set using a PropertyValue. These values will be used to construct or configure a component instance.

A PropertyValue should be included whenever the default property value is not correct in the given context.

UML base element(s) in the Profile and Stereotype

Constraint stereotyped as <PropertyValue>

Fully Scoped name

ECA::CCA::PropertyValue

Owned by

ComponentUsage

Extends

None

Properties

value

An expression for the value of the property.

UML Representation

Constraint::body

Related elements

Owner

The component usage being configured with a value.

UML Representation

ModelElement::namespace

Fills

The property being modified.

UML Representation

Constraint:constrainedElement referencing an attribute of <Owner>.

Constraints

- "fills" must relate to a property definition of the ProcessComponent that the owner uses.
- The type returned by the PropertyValue expression must be compatible with the type defined by the PropertyDefinition.

2.2.3.6 ContextualBinding

Semantics

A composition is able to use abstract ProcessComponents in compositions – we call these abstract compositions. The use of an abstract composition implies that at some point a concrete component will be bound to that composition. That binding may be done at runtime or when the composition is used as a component in another composition.

For example, a composed “Pricing” component may use an abstract component “PriceFormula”. In our “InternationalSales” composition we may want to say that “PriceFormula” uses “InternationalPricing”.

Contextual Binding allows the substitution of a more concrete ProcessComponent for a compatible abstract ProcessComponent when an abstract composed ProcessComponent is used. So within the composition that uses the abstract component (International Sales) we say the use of a particular Component (use of PriceFormula) will be bound to a concrete component (InternationalPricing). These semantics correspond with the three relations out of ContextualBinding.

Note that other forms of binding may be used, including runtime binding. But these are out of scope for CCA. Some specializations of CCA may subtype ContextualBinding and apply selection formula to the binding, as is common in workflow systems.

An abstract composition may also be thought of as a pattern, with contextual binding being the parameter substitution.

UML base element(s) in the Profile and Stereotype

Binding stereotyped as <ContextualBinding>

Fully Scoped name

ECA::CCA::ContextualBinding

Owned by

Composition

Extends

None

Properties

None

Related elements

owner

The composition which is using the abstract composed component and wants to bind a more specific ProcessComponent for an abstract one. The owner of the ContextualBinding.

UML Representation

ModelElement::namespace

fills

The ComponentUsage which should have the ProcessComponent it uses replaced. This component usage does not have to be within the same composition as the contextual binding, it may be anywhere the component usage occurs visible from the scope of the composition owning the binding.

UML Representation

Binding::client

bindsTo

The concrete component which will be bound to the component usage.

UML Representation

Binding::supplier

Constraints

The ProcessComponent related to by "bindsTo" must be a subtype of the component used by the component usage related to by "fills".

2.2.3.7 CommunityProcess

Semantics

Community processes may be thought of as the "top level composition" in a CCA specification, it is a specification of a composition of ProcessComponents that work together for some purpose other than specifying another ProcessComponent.

One kind of CommunityProcess would be a business process, in which case the nested components represent business partner roles in that process. For example, a community process could define the usage of a buyer, a seller, a freight forwarder and two banks for a sale and delivery process.

Note that designs can be done "top down" or as an assembly of existing ProcessComponents (bottom up). When design is being done top down, it is usually the CommunityProcess which comes first and then ProcessComponents specified to fill the roles of that process.

CommunityProcesses are also useful for standards bodies to specify the roles and interactions of a B2B process.

UML base element(s) in the Profile and Stereotype

Subsystem stereotyped as <<CommunityProcess>> with a Composition

Fully Scoped name

ECA::CCA::CommunityProcess

Owned by

Package

Extends

Composition and Package

Properties

None

Related elements

None

Constraints

None

2.2.4 Document Model

The document model defines the information that can be transferred between and manipulated by ProcessComponents. It also forms the base for information in entities.

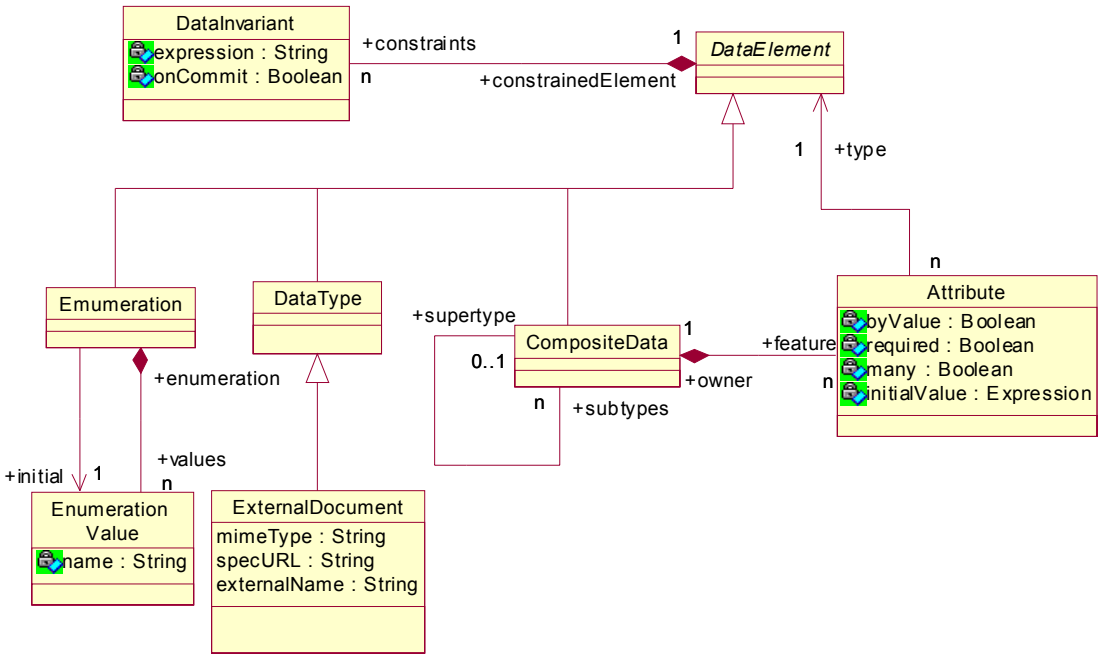

Figure 11: Document Metamodel

A **data element** represents a type of data which may either be primitive **DataTypes** or composite**. CompositeData** has named attributes which reference other types. Any type may have a **DataInvariant** expression.

Attributes may be **isByValue**, which are strongly contained or may simply reference other data elements provided by some external service. Attributes may also be marked as **required** and/or **many** to indicate cardinality. **DataTypes** define local data – these types are defined outside of CCA. **ExternalDocument** defines a document defined in an external type system. An **enumeration** defines a type with a fixed set of values

2.2.4.1 DataElement

Semantics

DataElement is the abstract supertype of all data types. It defines some kind of information.

UML base element(s) in the Profile and Stereotype

Classifier (no stereotype)

Fully Scoped name

ECA::DocumentModel::DataElement

Owned by

Package

Extends

PackageContent

Properties

None

Related elements

constraints

Constraints applied to the values of this data type.

Constraints

None

2.2.4.2 DataType

Semantics

A primitive data type, such as an integer, string, picture, movie…

Primitive data types may have their structure and semantics defined outside of CCA. The following data types are defined for all specializations of CCA: String, Integer, Float, Decimal, Boolean.

UML base element(s) in the Profile and Stereotype

DataType (no stereotype)

Fully Scoped name

ECA::DocumentModel::DataType

Owned by

Package

Extends

DataElement

Properties

None

Related elements

None

Constraints

None

2.2.4.3 Enumeration

Semantics

An enumeration defines a type that may have a fixed set of values.

UML base element(s) in the Profile and Stereotype

Corresponds to User defined enumeration stereotypes of UML DataType.

Fully Scoped name

ECA::Documentmodel::Enumeration

Owned by

Package

Extends

DataElement

Properties

None

Related elements

Values

The set of values the enumeration may have.

UML Representation

ModelElement::namespace

Initial

The initial, or default, value of the enumeration.

UML Representation

Tagged value

Constraints

None

2.2.4.4 EnumerationValue

Semantics

A possible value of an enumeration.

UML base element(s) in the Profile and Stereotype

The values of User defined enumeration stereotypes of UML DataType.

Fully Scoped name

ECA::DOCUMENTMODEL::EnumerationValue

Owned by

Enumeration

Extends

None

Properties

name

Related elements

Enumeration

The owning enumeration.

UML Representation

ModelElement:namespace

Constraints

None

2.2.4.5 CompositeData

Semantics

A datatype composed of other types in the form of attributes.

UML base element(s) in the Profile and Stereotype

Class Stereotyped as <<CompositeData>>

Fully Scoped name

ECA::DocumentModel::CompositreData

Owned by

Package

Extend

DataElements

Properties

None

Related elements

Feature

The attributes which form the composite.

UML Representation

Classifier.feature

Supertype

A type from which this type is specialized. The composite will include all attributes of all supertypes as attributes of itself.

Subtypes

The types derived from this type.Constraints

UML Representation

Generalization

2.2.4.6 Attribute

Semantics

Defines one “slot” of a composite type that may be filled by a data element of “type”.

UML base element(s) in the Profile and Stereotype

Attribute (No stereotype)

Fully Scoped name

ECA::DOCUMENTMODEL::Attribute

Owned by

CompositeData

Extends

None

Properties

isByValue

Indicates that the composite data is stored within the composite as opposed to referenced by the composite.

UML Representation

Stand-alone Tagged Value to apply to UML Attribute (a Stereotype of Attrbute is not created to hold this TaggedValue :

required

Indicates that the attribute slot must have a value for the composite to be valid.

UML Representation

StructuralFeature::multiplicity

many

Indicates that there may be multiple occurrences of values. These values are always ordered.

UML Representation

StructuralFeature::multiplicity

initialValue

An expression returning the initial value of the attribute.

UML Representation

Attribute::initialValue

Related elements

type

The type of information which the attribute may hold. Type instances may also be filled by a subtype.

UML Representation

StructuralFeature::type

owner

The composite of which this is an attribute.

UML Representation

ModelElement::namespace

Constraints

None

2.2.4.7 DataInvariant

Semantics

A constraint on the legal values of a data element.

UML base element(s) in the Profile and Stereotype

Constraint

Fully Scoped name

ECA::DOCUMENTMODEL::DataInvarient

Owned by

DataElement

Extends

None

Properties

Expression

The expression which must return true for the data element to be valid.

UML Representation

Constraint::body

isOnCommit (Default: False)

True indicates that the constraint only applies to a fully formed data element, not to one under construction.

UML Representation

Tagged Value

Related elements

ConstrainedElement

The data element that will be constrained.

UML Representation

Constraint::constrainedElement

2.2.4.8 ExternalDocument

Semantics

A large, self contained document defined in an external type systems such as XML, Cobol or Java that may or may not map to the ECA document model.

UML base element(s) in the Profile and Stereotype

DataType Stereotyped as <<ExternalDocument>>

Fully Scoped name

ECA::DOCUMENTMODEL::ExternalDocument

Owned by

Package

Extends

DataElement

Properties

All properties are tagged values

MimeType

The type of the document specified as a string compatible with the “mime” declarations.

SpecURL

A reference to an external document definition compatible with the mimiType, such as a DTD or Schema. If the MimeType does not define a specification form (E.G. GIF) then this attribute will be blank.

ExternalName

The name of the document within the SpecURL. For example, an element name within a DTD. If the MimeType does not define a specification form (E.G. GIF) or the specification form only specifies one document then this attribute will be blank.

Related elements

None

Constraints

None

2.2.5 Model Management

Model management defines how CCA models are structured and organized. It directly maps to its UML counterparts and is only included as an ownership anchor for the other elements.

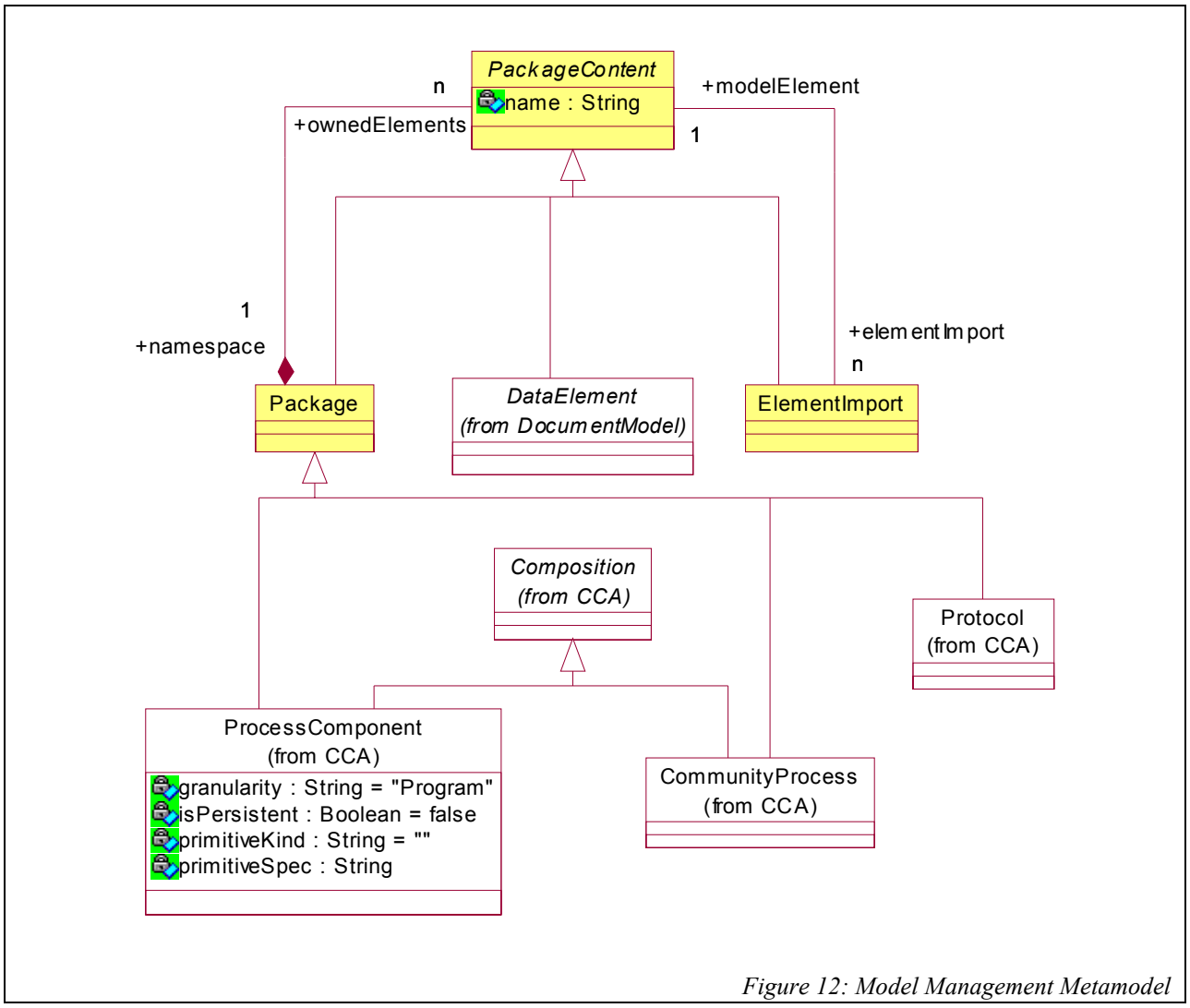

Figure 12: Model Management Metamodel

A **package** defines a logical hierarchy of reusable model elements. Elements that may be defined in a package are **PackageContent** and may be ProcessComponents, Protocols, DataElements, CommunityProcesses and other packages. A **ImportedElement** defines a "shortcut" visibility of a package content in a package that is not its owner. Shortcuts are useful to organize reusable elements from different perspectives.

Note that ProcessComponents are also packages, allowing elements which are specific to that component to be defined within the scope of that component.

2.2.5.1 Package

Semantics

Defines a structural container for “top level” model elements that may be referenced by name for other model elements.

UML base element(s) in the Profile and Stereotype

Package

Fully Scoped name

ECA::ModelManagement::Package

Owned by

Package or model (global scope)

Extends

PackageContent

Properties

None

Related elements

OwnedElements

The model elements within the package and visible from outside of the package.

UML Representation

Namespace::OwnedElement

Constraints

None

2.2.5.2 PackageContent

Semantics

An abstract capability that represents an element that may be placed in a package and thus referenced by name from any other element.

UML base element(s) in the Profile and Stereotype

ModelElement

Fully Scoped name

ECA::ModelManagement::

Owned by

Package

Extends

None

Properties

name

UML Representation

ModelElement::name

Related elements

namespace

UML Representation

ModelElement::namespace

Constraints

2.2.5.3 ElementImport

Semantics

Defines an “Alias” for one element within another package.

UML base element(s) in the Profile and Stereotype

ElementImport (No Stereotype)

Fully Scoped name

ECA::ModelManagement::ElementImport

Owned by

Package

Extends

PackageContent

Properties

None

Related elements

ModelElement

The element to be imported.

Constraints

None

2.3 CCA Notation

CCA uses UML notation with a few extensions and conventions to make diagrams more readable and compact for CCA aware tools. The UML mapping shown how CCA is expressed in the UML Metamodel which has standard notation. Unless stated otherwise, all other UML elements use the base UML 1.4 notation. The following are additions this base UML 1.4 notation.

2.3.1 CCA Specification Notation

A ProcessComponent is based on the notation for a subsystem with extensions for ports and properties. Consider the following diagram template for ProcessComponent notation.

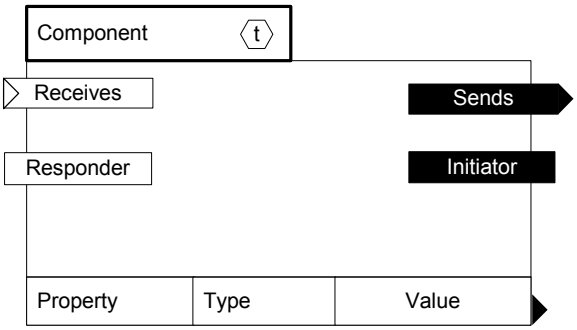

Figure 13: ProcessComponent specification notation

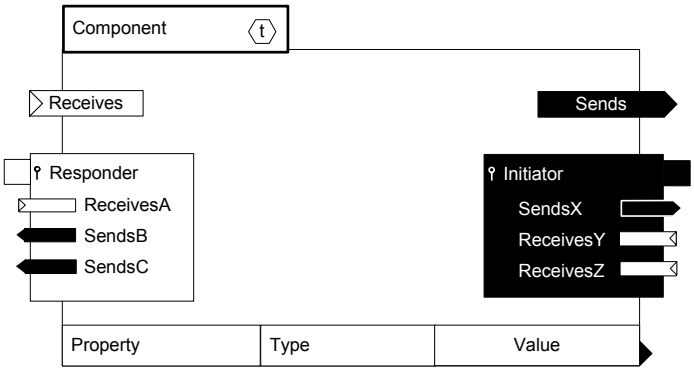

Figure 14: ProcessComponent specification notation (expanded ProtocolPorts)

- A **ProcessComponent** represents its external contract as a subsystems with the following addition:
- The ProcessComponent **type** may be represented as an icon in the component name compartment. "t" above.
- **Ports** are represented as going through the boundary of the box. The port is itself a smaller rectangle with the name of the port inside the rectangle.. In the above, "Receives", "Sends", "Responder" and "Initiator" are all ports. The type of the port is not represented in the diagram.
- **Flow ports** are represented as an arrow going through a box. Flow ports that send have the arrow pointing out of the box while flow ports that receive (Receives) have an arrow pointing into the box. A sender has the background and text color inverted.
- **Protocol ports** and **Operation ports** are boxes extending out of the component. Protocol ports representing an initiator have the colors of their background and text

reversed. In the above, “Initiator” is a protocol port of an initiator and “Responder” is a protocol port that is not an initiator. ProtocolPorts may show nested, the Ports of the used Protocol.

- **Multiports** are shown as a shaded box grouping the set of ports it contains.
- **Property Definitions** are in a separate compartment listing the property name, type and default value (if any). The name, type and value are separated by lines. Each property is on a separate line.

2.3.2 Composite Component Notation

A composite is shown as a ProcessComponent with the composition in the center. The composition is a new notation but may also be rendered with a UML collaboration.

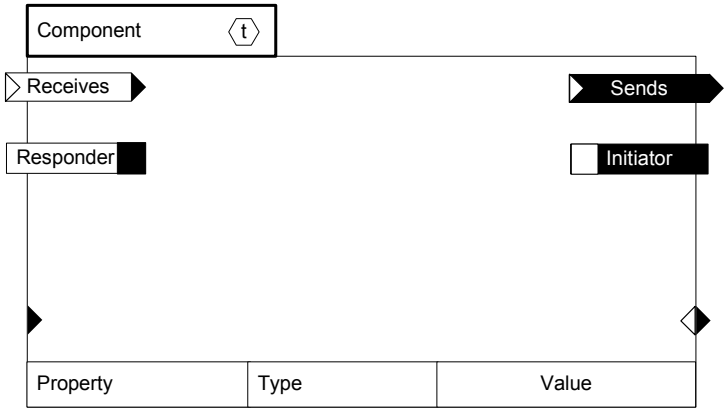

Figure 15: Composite Component notation (without internal ComponentUsages)

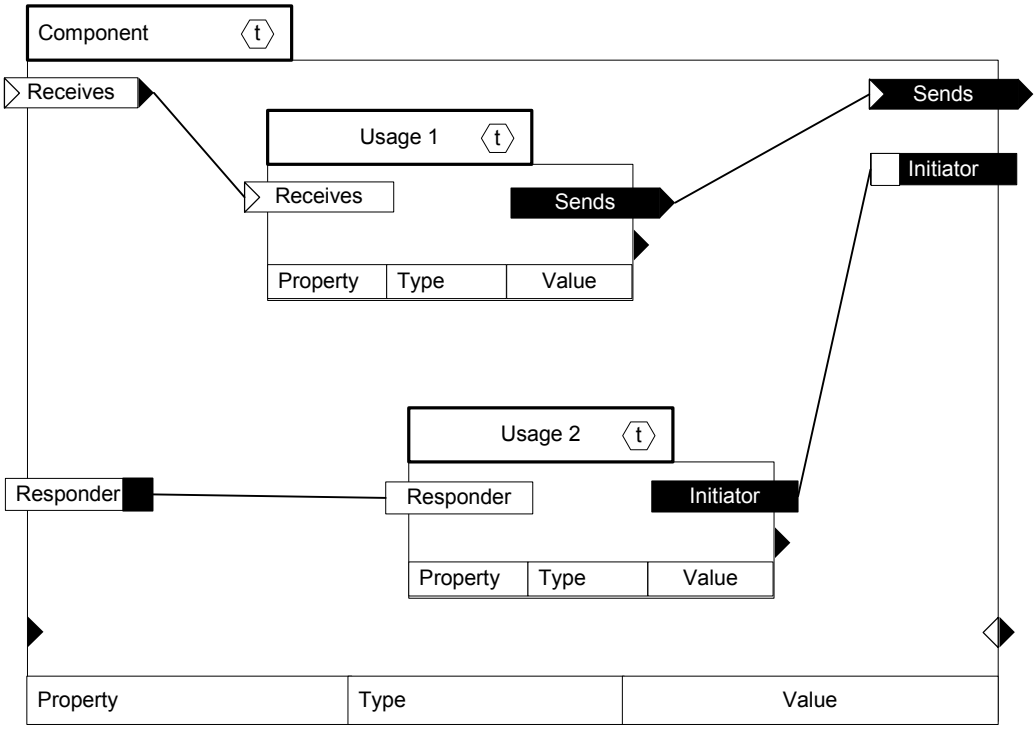

Figure 16: Composite Component notation

- The **ports** on the composite component being defined are shown in the same way as they are on a ProcessComponent, but in this case represent the **port connector.**
- A **component usage** is shown as a smaller version of a ProcessComponent inside the composite component. Note Usage (1..2) are component usages.
- **Port connectos** are shown in the same fashion as ports, on component usages. The ports on Usage 1..2 are all port usages.
- **Connectors** are shown as lines between port usages or port proxies. All the lines in the above are connectors.
- **Property values** may be shown on component usages (in the same way as the property definition), or may be suppressed.

2.3.3 Community Process Notation

A community process is shown in the same way as a composite component with the exception that a community process has no external ports.

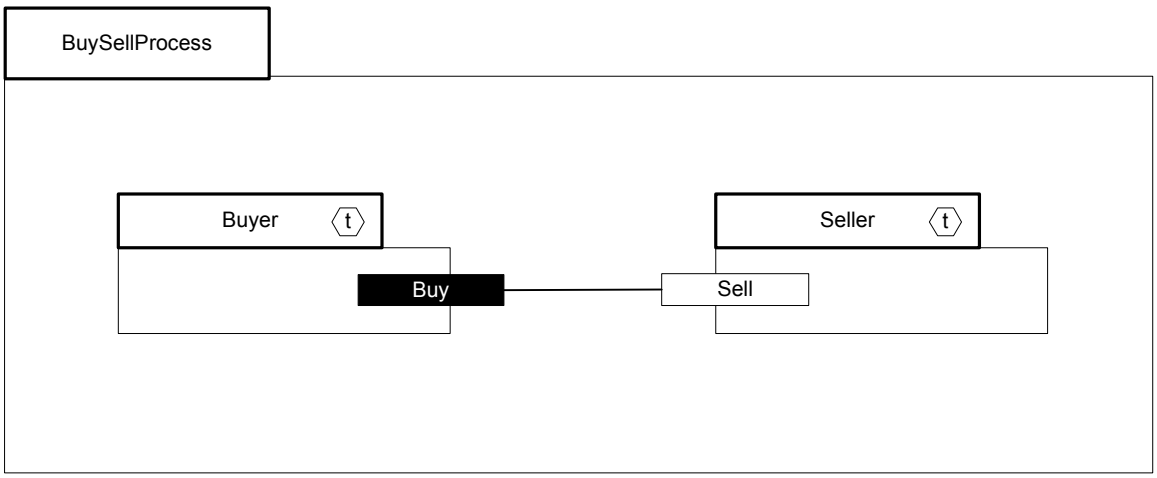

Figure 17: CommunityProcess notation

In the above example "BuySellProcess" is a community process with component usage for "Buyer" and "Seller" which are connected via their "buy" and "sell" ports, respectively.

2.4 UML Profile

The CCA profile specifies how CCA concepts relate to and are represented in standard UML using stereotypes, tagged values and constraints. This allows off-the-shelf UML tools to represent CCA and interchange CCA models.

The CCA profile is organized as a single package which corresponds to the ECA::CCA package in the logical model and the CCA <<profile>> package. In addition there is a package for the document model which is used by CCA.

2.4.1 Tables mapping concepts to profile elements

The following tables provide a summary of the CCA elements as stereotypes and tagged values. These stereotypes and tagged values may be used in standard UML models, and represented in standard UML diagrams (See 2.5"Diagramming CCA" for an example).

Metamodel element name	**Stereotype name**	**UML base Class**	**Parent**	**Tags**	**Constraints**
ProcessComponent	ProcessComponent	Classifier	N/A	granularity isPersistent primitiveKind primitiveSpec	
Port	Port	Class	N/A	isSynchronous isTransactional direction postCondition	
FlowPort	FlowPort	Class	Port	typeProperty	

Metamodel element name	Stereotype name	UML base Class	Parent	Tags	Constraints
ProtocolPort	ProtocolPort	Class	Port	uses	
MultiPort	MultiPort	Class	Port		
OperationPort	N/A	Operation	Port		
Protocol	Protocol	Class	N/A		
Interface	N/A	Classifier	N/A		
InitiatingRole	InitiatingRole	Class	N/A		
RespondingRole	InitiatingRole	Class	N/A		
PropertyDefinition	PropertyDefinition	Attribute	N/A		
«enumeration» DirectionKind	DirectionKind	Enumeration			
«enumeration» GranularityKind	GranularityKind	Enumeration	N/A		
Direction (value)	initiates	Association	N/A		
Direction (value)	responds	Association	N/A		

Table 2: Stereotypes for Structural Specification (UML notation: Class Diagram)

Metamodel attribute name	Tag	Stereotype	Type	Multiplicity	Description
granularity	granularity	ProcessComponent	«enumeration» GranularityKind	1	
primitiveKind	primitiveKind		String	0..1	
primitiveSpec	primitiveSpec		String	0..1	
isPersistent	isPersistent		Boolean	1	default=false
isSynchronous	isSynchronous	Port and specializations: ProtocolPort or FlowPort or MultiPort or OperationPort	Boolean	1	default=false
isTransactional	isTransactional		Boolean	1	default=false
direction	direction		«enumeration» DirectionKind	1	
postCondition	postCondition		«enumeration» Status	0..1	
typeProperty	typeProperty	FlowPort	Attribute	0..1	Reference a PropertyDefinition of the owner ProcessComponent.

Table 3: TaggedValues for Structural Specification

Metamodel element name	Stereotype name	UML Base Class	Parent	Tags	Constraints
PortActivity	PortActivity	CompositeState	N/A	represents	
Transition	N/A (UML element)	Transition	N/A		
Pseudostate	N/A (UML element) or Success or Failure	Pseudostate	N/A		
Pseudostate	Success	FinalState	N/A		
Pseudostate	Failure	FinalState	N/A		
«enumeration» Status	Status	Enumeration			

Table 4: Stereotypes for Choreography (UML notation: Statechart Diagram)

Metamodel attribute name	Tag	Stereotype	Type	Multi plicity	Description
represents	represents	PortActivity	Class, constrained to «ProtocolPort» or «FlowPort» or «MultiPort» or «OperationPort»	1	

Table 5: TaggedValues for Choreography

Metamodel element name	Stereotype name	UML Base Class	Parent	Tags	Constr aints
Composition	Composition	Collaboration	N/A		
ComponentUsage	ComponentUsage	ClassifierRole	N/A		
PortConnector	PortConnector	ClassifierRole	N/A		
Connection	Connection	AssociationRole	N/A		
PropertyValue	PropertyValue	Constraint	N/A		
ContextualBinding	ContextualBinding	Binding	N/A		
CommunityProcess	CommunityProcess	Subsystem	N/A		

Table 6: Stereotypes for Composition (UML notation: Collaboration Diagram at specification level)

Metamodel attribute name	Tag	Stereotype	Type	Multi plicity	Description
represents	represents	PortConnector	Class, constrained to «ProtocolPort» or «FlowPort» or «MultiPort»	1	

Table 7: TaggedValues for Composition

Metamodel element name	Stereotype name	UML Base Class	Parent	Tags	Constraints
CompositeData	CompositeData	Class	N/A		
ExternalDocument	ExternalDocument	DataType	N/A		
DataInvariant	DataInvariant	Constraint	N/A		
DataType	N/A (UML)	DataType	N/A		
Enumeration	N/A (UML)	Enumeration	N/A		
Attribute	N/A (UML)	Attribute	N/A		

Table 8: Stereotypes for DocumentModel (UML notation: Class Diagram)

Metamodel attribute name	Tag	Stereotype	Type	Multi plicit y	Description
isOnCommit	isOnCommit	DataInvariant	Boolean	1	
isByValue	isByValue	N/A		1	Apply to Attribute of

Metamodel attribute name	Tag	Stereotype	Type	Multiplicity	Description
					«CompositeData»
mimeType	mimeType	ExternalDocument	String	0..1	
specURL	specURL		String	0..1	
externalName	externalName		String	0..1	

Table 9: TaggedValues for DocumentModel

2.4.2 Introduction

The UML Profile for CCA accesses a number of UML Packages. The CCA <<profile>> extends these packages with CCA stereotypes & semantics.

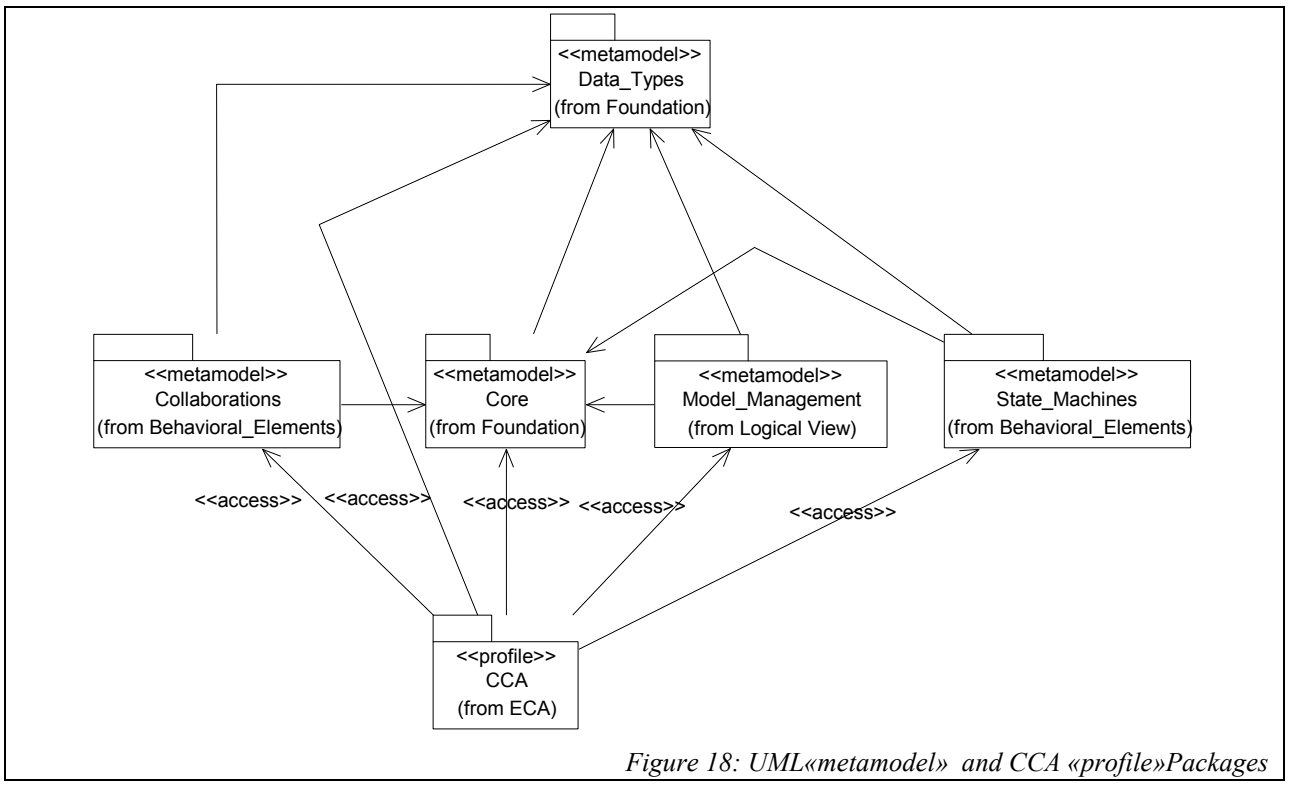

Figure 18: UML«metamodel» and CCA «profile»Packages

Each CCA stereotype extends a specific UML model element as shown below.

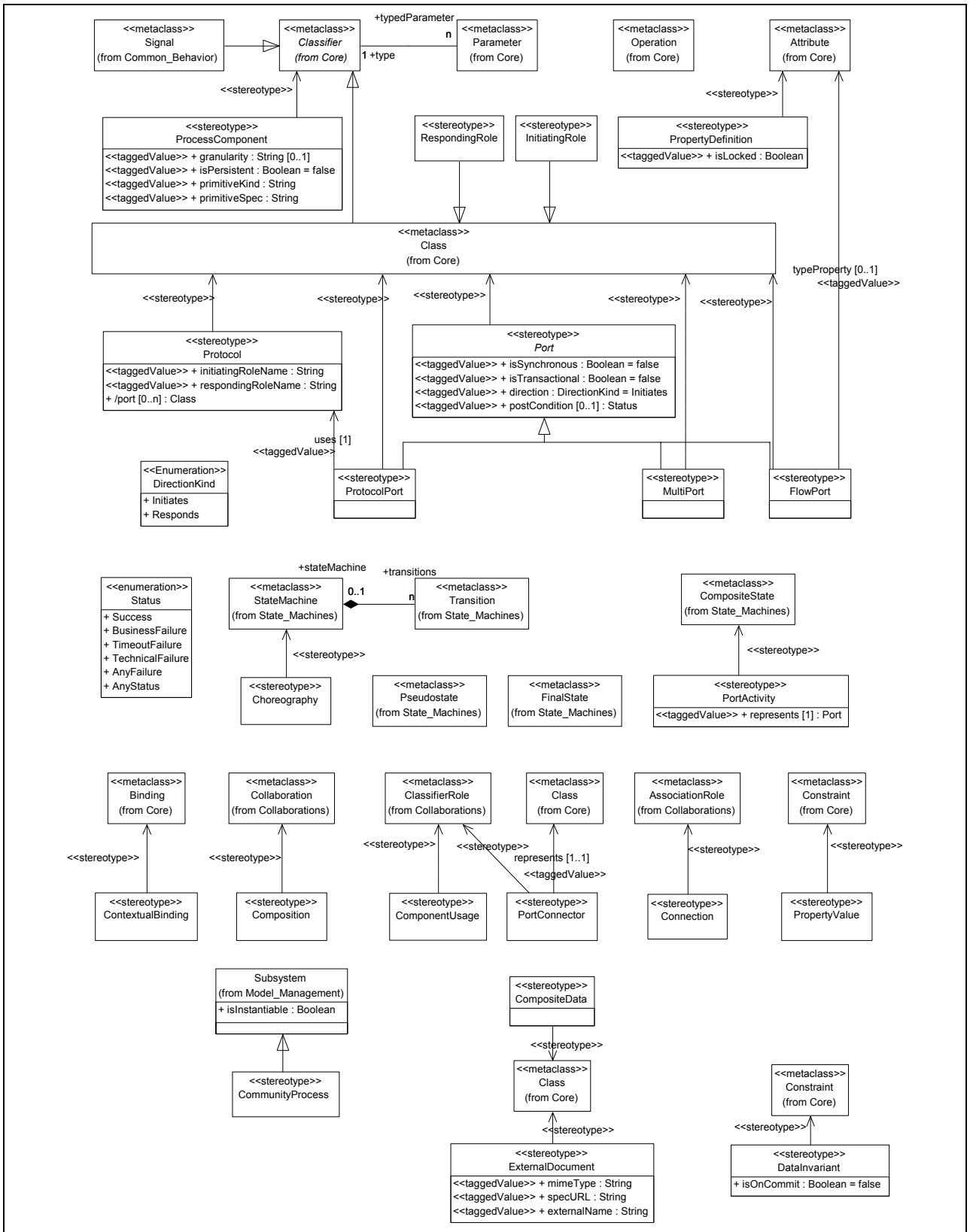

Figure 19: Stereotypes in the UML Profile for CCA

2.4.3 Stereotypes for Structural Specification

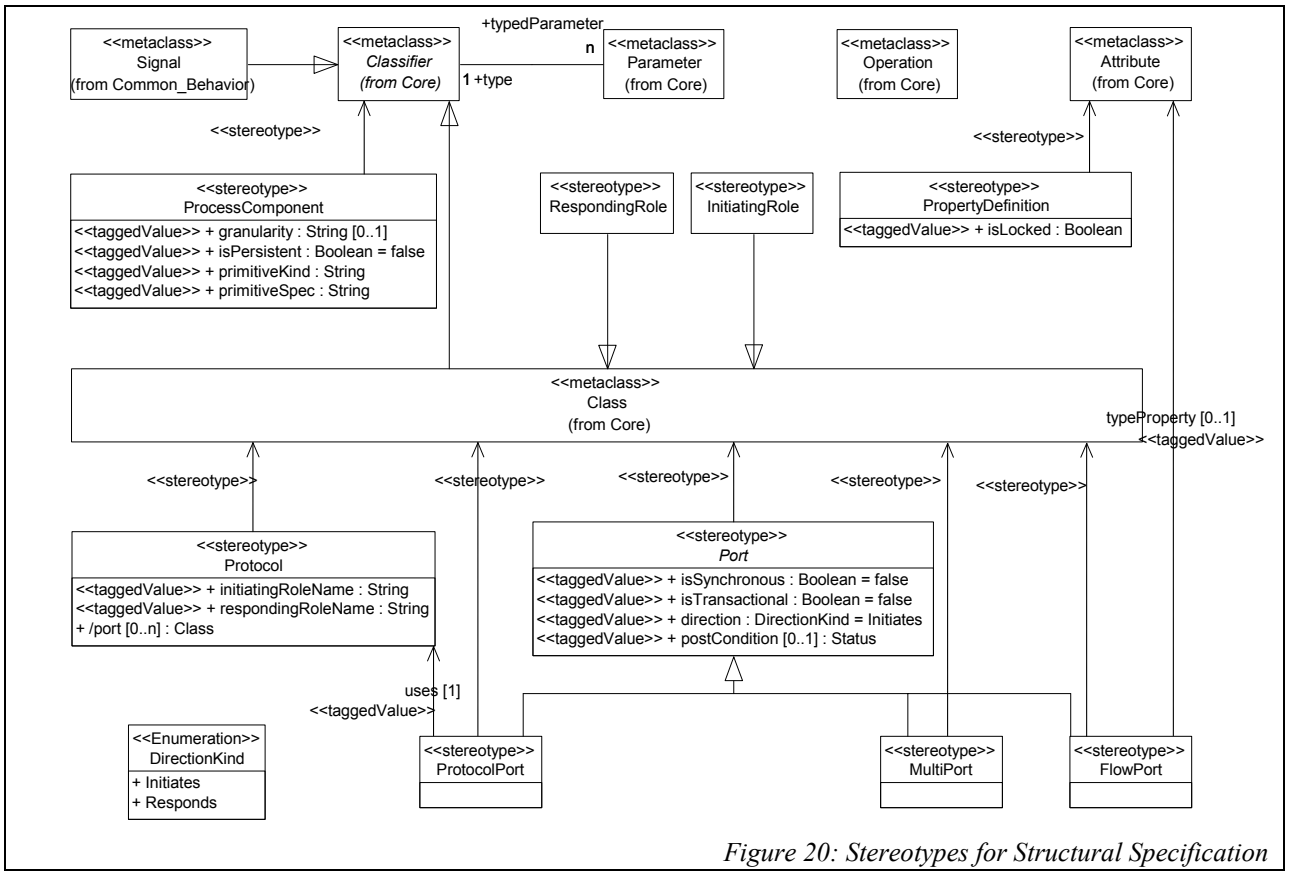

Figure 20: Stereotypes for Structural Specification

Applicable Subset

Classifier, Class, Attribute

2.4.3.1 «ProcessComponent»

Inheritance

Foundation::Core::Classifier
«ProcessComponent»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships[3]

Relationship	Role(s)
Ports	owner
Generalization	supertype subtypes {only with «ProcessComponent»}
Properties	component
Uses	owner
ComponentUsages	owner
Bindings	owner
Bindings	bindsTo
Connections	_connections
Nodes	_nodes
PortUsages	extent
Is_A_Choreography	is_specialization
Is_A_Composition	is_ specialization
PackageElements owner	ownerElements
ImportElement modelElement	elementImport

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

Tagged Value name	Type	Multiplicity	Description
granularity	String	0..1	
primitiveKind	String	0..1	
primitiveSpec	String	0..1	
isPersistent	Boolean	1	default=false

[3] The “Relationships” header references the relationships in which the Model Element participates, and the name of the role in the relationship. The section "Relationships", see 2.4.8 below, includes the specifications for these relationships, and their mapping between metamodel and UML representation.

Constraints expressed generically

The set of all the «Port» of a «ProcessComponent» is the set of «Port» or its specializations, that are aggregated in the «ProcessComponent».

The supertype of a «ProcessComponent» must be a «ProcessComponent».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context ProcessComponent

inv:
   supertype->isEmpty() or
   supertype.isStereoKinded("ProcessComponent")

def:
  -- the Ports in the ProcessComponent :
  -- composed in the ProcessComponent

 let ports : Set( Class) =
  (association->select( anAssociationEnd : AssociationEnd |
   anAssociationEnd.aggregationKind = ak_composite)
  ->association->connection – association)
  ->participant
  ->select( aClassifier : Classifier|
   anElement.isStereoKinded( «Port»))
```

Diagram Notation

N/A

2.4.3.2 «Port»

Inheritance

Foundation::Core::Class
«Port»

Instantiation in a model

Abstract

Semantics

Corresponds to the element of same name in the metamodel.

The «Port» stereotype has been introduced for clarity and brevity, defining in a common ancestor, the taggedValues corresponding to attributes of Port in the metamodel, and reused along the stereotypes specialization of «Port» : «FlowPort», «ProtocolPort», «MultiPort» and «OperationPort».

Relationships

Relationship	Role(s)
Ports	ports
Represents	represents

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

Tagged Value name	Type	Multiplicity	Description
isSynchronous	Boolean	1	default=false
isTransactional	Boolean	1	default=false
direction	DirectionKind	1	
postCondition	«enumeration» Status	0..1	

Constraints expressed generically

A «Port» must be aggregated into a «Protocol» or a «ProcessComponent», or a «MultiPort».

Note that the metamodel Interface corresponds in the UML Profile to a UML Classifier which may or may not by a UML Interface, and that the metamodel OperationPort corresponds to a UML Operation. However, UML Interface is the recommended model element to use. Although in the metamodel both Interface and OperationPort may contain other Port, in the UML Profile these, and their relationships are directly supported by UML. Neither Interface or OperationPort appear in the constraint below, as candidate owners for «Port». This allows arbitrary UML classifiers (of any kind) to be used with CCA. Only the operations of these classifiers will correspond to CCA elements.

The relationship between the Port and the PortOwner shall have the stereotype <<initiates>> or the stereotype <<responds>> which shall have the same value as "direction".

Formal Constraints Expressed in Terms of the UML Metamodel

```
context Port

inv:
  aggregatedOwner->notEmpty()

inv:
  ownerAggregation.isStereoKinded("initiates") implies
     direction = "Initiates"
```

```
inv:
  ownerAggregation.isStereoKinded("responds") implies
     direction = "Responds"

def:
  -- the owner of the Port
 let aggregatedOwner : Class = ownerAggregation.participant

def:
  let ownerAggregation : Class =
  (association->association->connection – association)->
     select( anAssociationEnd : AssociationEnd |
     anAssociationEnd.aggregationKind = ak_composite)
  ->select( anAssocRole : AssociationRole|
   anAssocRole->participant.isStereoKinded( «Protocol») or
     anAssocRole->participant.isStereoKinded(
  «ProcessComponent») or
     anAssocRole->participant..isStereoKinded(
  «MultiPort»))
  ->any( true)
```

Diagram Notation

N/A

2.4.3.3 «FlowPort»

Inheritance

Foundation::Core::Class
ECA::CCA::ComponentSpecification::«Port»
«FlowPort»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	**Role(s)**
FlowType	_type
TypeProperty	constrains

Tagged Values

Tagged Value name	Type	Multiplicity	Description
typeProperty	Attribute	0..1	Refer to a «PropertyDefinition» of the owner «ProcessComponent». When the «ProcessComponent» is used as a «ComponentUsage», the value held by the «PropertyValue» in the «ComponentUsage» will be interpreted as the actual type of the «FlowPort», for its specific «PortUsage» in the «ComponentUsage».

Constraints expressed generically

The «FlowPort» must reference as its type a DataType, Enumeration, «CompositeData» or «ExternalDocument» or their specializations.

The typeProperty of «FlowPort», if is specified, it must reference an Attribute stereotyped as «PropertyDefinition», owned by the same «ProcessComponent» that owns the «FlowPort». If the initialValue of the «ProperyDefinition» is set, then the value must be the name of a DataElement, Enumeration, «CompositeData» or «ExternalDocument».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context FlowPort

inv:
  type->notEmpty()


inv:
  typeProperty->isEmpty() or (
    typeProperty.owner = this.aggregatedOwner)

def:
  let type : Classifier =
  (association->association->connection - association)-
  >participant
    ->select( aClassifier : Classifier|
          anElement.isOclKindOf( DataElement) or
          anElement.isOclKindOf( Enumeration) or
    anElement.isStereoKinded( «CompositeData») or
    anElement.isStereoKinded( «ExternalDocument»))
```

Diagram Notation

N/A

2.4.3.4 «ProtocolPort»

Inheritance

Foundation::Core::Class
ECA::CCA::ComponentSpecification::«Port»
«ProtocolPort»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
ProtocolType	_uses

Tagged Values

N/A

Constraints expressed generically

A «ProtocolPort» must reference a «Protocol», or its specializations, through a Generalization Relationship, with the «Protocol» as the parent.

Formal Constraints Expressed in Terms of the UML Metamodel

```
context ProtocolPort
inv:
  generalization->notEmpty() and
  generalization.parent->select( aGeneralizable :
  GeneralizableElement |
     aGeneralizable.isStereoKinded("Protocol"))
  ->notEmpty()
```

Diagram Notation

N/A

2.4.3.5 «MultiPort»

Inheritance

Foundation::Core::Class
ECA::CCA::ComponentSpecification::«Port»
«MultiPort»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Ports	owner

Tagged Values

N/A

Constraints expressed generically

All the «Port» aggregated by the «MultiPort», must be «FlowPort» or its specializations.

Formal Constraints Expressed in Terms of the UML Metamodel

```
context MultiPort

inv:
   ports->forAll( aClass : Class |
   aClass.isStereoKinded("FlowPort"))

def:
 let ports : Set( Class) =
  (association->select( anAssociationEnd : AssociationEnd |
    anAssociationEnd.aggregationKind = ak_composite)
  ->association->connection – association)
  ->participant
  ->select( aClassifier : Classifier|
   anElement.isStereoKinded( «Port»))
```

Diagram Notation

N/A

2.4.3.6 UML Operation represents OperationPort

Semantics

The concept of OperationPort in the metamodel, is represented by a standard UML operation.

The OperationPort is constrained to contain only FlowPorts.

The signature, of the UML Operation representing an OperationPort, is derived from the type of the one and only FlowPort of the OperationPort, with direction="initiates". For each Attribute of the FlowPort, the UML Operation will have an input Parameter with type equal to the type of the Attribute in the FlowPort.

For each ownedFlowPort with direction="responds" and postCondition="Success", then the UML Operation will have return Parameters with same type as the type of the FlowPort.

All other FlowPort in the OperationPort with direction="responds", correspond to raisedException Signal of the UML Operation. The structure of the Signal is derived from the FlowPort type : the Signal will have Attribute with same name and type of the Attribute of the type of the FlowPort.

Relationships

N/A

Tagged Values

N/A

Constraints expressed generically

.N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.3.7 «Protocol»

Inheritance

Foundation::Core::Class
«Protocol»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Ports	owner
ProtocolType	_uses
Generalization	supertype subtypes (only with «Protocol»)
Node	nodes
Connection	connections
PackageElements	owner ownedElements
Is_a_Choreography	is_specialization
ImportElement	modelElement elementImport
Initiator	_initiator
Responder	_responder

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

The supertype of a «Protocol» must be a «Protocol».

The set of all the «Port»s of a «Protocol» is the set of «Port»s or its specializations, that are aggregated in the «Protocol».

A «Protocol» may have an Aggregation with at most one «InitiatingRole».

A «Protocol» may have an Aggregation with at most one «RespondingRole».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context Protocol

inv: initiatingRole->size() < 2

inv: repondingRole->size() < 2

inv:
   supertype->isEmpty() or
   supertype.isStereoKinded("Protocol")

def:
  -- the Ports in the Protocol : Association composed in
   the Protocol

 let ports : Set( Class) =
  (association->select( anAssociationEnd : AssociationEnd |
    anAssociationEnd.aggregationKind = ak_composite)
  ->association->connection – association)
  ->participant
  ->select( aClassifier : Classifier|
   anElement.isStereoKinded( «Port»))

def:
   let initiatingRole : Class = (association->select(
   anAssociationEnd : AssociationEnd |
     anAssociationEnd.aggregationKind = ak_composite)
  ->association->connection – association)
  ->participant
  ->select( aClassifier : Classifier|
   anElement.isStereoKinded( «InitiatingRole»))

def:
   let repondingRole: Class = (association->select(
   anAssociationEnd : AssociationEnd |
     anAssociationEnd.aggregationKind = ak_composite)
  ->association->connection – association)
  ->participant
  ->select( aClassifier : Classifier|
   anElement.isStereoKinded( «RespondingRole»))
```

Diagram Notation

N/A

2.4.3.8 «InitiatingRole»

Inheritance

Foundation::Core::Class

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Initiator	–

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

context InitiatingRole

Diagram Notation

N/A

2.4.3.9 «RespondingRole»

Inheritance

Foundation::Core::Class
«RespondingRole»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Responder	_responder

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

context RespondingRole

Diagram Notation

N/A

2.4.3.10 UML Classifier represents Interface

Inheritance

N/A

Instantiation in a model

Concrete subtypes of classifier.

Semantics

The metamodel element Interface corresponds to the UML Classifier.

Foundation::Core::Classifier

A metamodel Interface can only contain metamodel OperationPort, and OperationPort can only contain constrained FlowPort.

An Classifier Classifier contains UML Operation features, corresponding to the OperationPort of the metamodel Interface.

The metamodel FlowPort, owned by OperationPort, are mapped into the UML Parameter of the UML Operation. Parameter include the return type, and alternate exceptional result types.

The metamodel FlowPort of the OperationPort must comply with constraints, ensuring that the OperationPort FlowPort can be mapped to the Parameter of the UML Operation.

The metamodel Interface can only have OperationPort and FlowPort, because only these can be mapped to UML Operation. The OperationPort and FlowPort of Interface, can only have direction="responds".

The «InitiatingRole», initiator of the Classifier, is the role that invokes operations in the Classifier. The «RespondingRole», responder of the Classifier, is the role that implements the operations in the Classifier.

Relationships

Relationship	**Role(s)**
ProtocolType	_uses
Generalization	supertype subtypes (only with Classifier)
Node	nodes
Connection	connections
PackageElements	owner ownedElements
Is_a_Choreography	is_specialization
Initiator	_initiator
Responder	_responder

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	**UML attribute name**	**UML attribute owner**	**Description**
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.3.11 «PropertyDefinition»

Inheritance

Foundation::Core::Attribute
«PropertyDefinition»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	**Role(s)**
Properties	properties
PropertyType	type
TypeProperty	typeProperty
ValueFor	fills

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	**UML attribute name**	**UML attribute owner**	**Description**
name	name	ModelElement	
initial	initialValue	Attribute	
isLocked	changeability	StructuralFeature	

Tagged Values

N/A

Constraints expressed generically

The owner of an Attribute stereotyped «PropertyDefinition» must be stereotyped as «ProcessComponent» or its specializations.

The type of an Attribute stereotyped «PropertyDefinition» must be set, and be a DataType, or an Enumeration, or a Class stereotyped as «CompositeData» or its specializations.

If the «PropertyDefinition» is the typeProperty of a «FlowPort», owned by the same «ProcessComponent» that owns the «PropertyDefinition», then if the initialValue of the «ProperyDefinition» is set, then the value must be the name of a DataElement, Enumeration, «CompositeData» or «ExternalDocument».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context PropertyDefinition

inv:
  owner->notEmpty() and
     owner.isStereoKinded( "ProcessComponent")

inv:
  type->notEmpty() and (
     type.oclIsTypeOf( DataType) or
     type.oclIsTypeOf( Enumeration) or
     type.isStereoKinded( "CompositeData"))

-- ojo constrain initialValue when typeProperty of a
  FlowPort
```

Diagram Notation

N/A

2.4.3.12 «enumeration» DirectionKind

Instantiation in a model

Concrete

Semantics

Corresponds to the enumeration named "DirectionType" in the metamodel.

The DirectionKind enumeration in the metamodel is a UML Enumeration.

Enumeration Literals

Corresponding to the enumeration literals of same name in the metamodel.

- Initiates
- Responds

2.4.3.13 «enumeration» GranularityKind

Instantiation in a model

Concrete

Semantics

Corresponds to the enumeration named “GranularityKind” in the Meta-model, used by the metaatribute named "granularity", of ProcessComponent.

The set of candidate values for "granularity" in the metamodel, has been formalized in the UML Profile as an Enumeration named "GranularityKind".

Specializations of CCA may define specializations of GranularityKind with additional EnumerationLiterals..

Enumeration Literals

Corresponding to the enumeration literals of same name and semantics, in the metamodel.

- Program
- Owned
- Shared

2.4.4 Stereotypes for Choreography

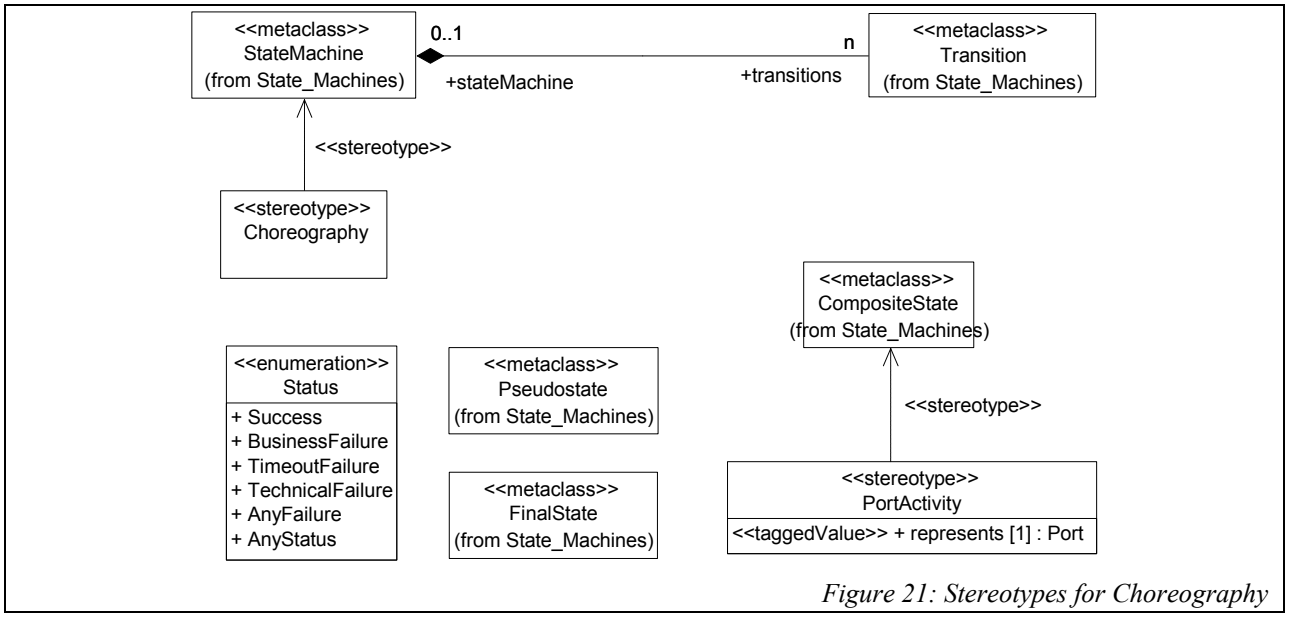

Figure 21: Stereotypes for Choreography

Applicable Subset

StateMachine, CompositeState, Transition, Pseudostate, FinalState

2.4.4.1 «Choreography»

Inheritance

Behavioral_Elements::State_Machines::StateMachine
«Choreography»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Is_a_Choreography	is_generalization
Nodes	_node
Connections	_connections

Tagged Values

N/A

Constraints expressed generically

The context of a StateMachine stereotyped as «Choreography» will be a Classifier stereotyped as «ProcessComponent» or a Class stereotyped as «Protocol» or a Subsystem stereotyped as <<CommunityProcess>>, or their specializations.

Formal Constraints Expressed in Terms of the UML Metamodel

```
context Choreography

inv:
  context->notEmpty() and (
     context->isStereoKinded( «ProcessComponent») or
     context->isStereoKinded( «Protocol») or
     context->isStereoKinded( «CommunityProcess»))
```

Diagram Notation

N/A

2.4.4.2 «PortActivity»

Inheritance

Behavioral_Elements::State_Machines::CompositeState
«PortActivity»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

When a PortActivity in the metamodel references as "represents" a FlowPort, then it corresponds to a «PortActivity» stereotype of CompositeState with no subvertex.

When the PortActivity in the metamodel references as "represents" a MultiPort, then it corresponds to a «PortActivity» stereotype of CompositeState with subvertexes «PortActivity» corresponding to the «FlowPort» of the «MultiPort».

When the PortActivity in the metamodel references as "represents" a «ProtocolPort», then it corresponds to a «PortActivity» stereotype of CompositeState.

To choreograph the «Port» in the "represents" «ProtocolPort», in the context of the «PortActivity», then «PortActivity» subvertexes can be nested, corresponding to the «Port» of the «Protocol» of the "represents" «ProtocolPort».

Relationships

Relationship	**Role(s)**
Nodes	nodes
Target	target
Source	source
PortUsages	portsUsed
Represents	_represents

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	**UML attribute name**	**UML attribute owner**	**Description**
name	name	ModelElement	Initialize equal to the name of the ""represents"" «Port»

Tagged Values

Tagged Value name	**Type**	**Multiplicity**	**Description**
represents	Class,	1	

Tagged Value name	Type	Multiplicity	Description
	constrained to «Port» or its specializations		

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

```
context PortActivity
```

Diagram Notation

N/A

2.4.4.3 UML Transition

Inheritance

N/A

Instantiation in a model

Concrete

Semantics

The metamodel element Transition corresponds to the UML model element of the same name.

Behavioral_Elements::State_Machines::Transition

The "preCondition" metaattribute corresponds to a UML Guard whose expression body will evaluate true under the same conditions as it would the "preCondition" metaattribute.

Relationships

Relationship	Role(s)
Target	incoming
Source	outgoing
Connections	connections

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.4.4 UML Pseudostate

Inheritance

N/A

Instantiation in a model

Concrete

Semantics

The metamodel element Pseudostate corresponds to the UML model element of the same name.

Behavioral_Elements::State_Machines:: Pseudostate

CCA Pseudostate mapps to UML Pseudostate except when the CCA-metamodel attribute "kind" of the Pseudostate has value "Success" or "Failure", that map to stereotypes of UML FinalState. Please see stereotypes «Success» and «Failure», below.

The semantics of the metamodel element Pseudostate are equivalent to the semantics of UML Pseudostate with corresponding "kind" values.

Metamodel kind	UML kind : Foundation::Data_Types::PseudostateKind
choice	pk_choice
fork	pk_fork
initial	pk_initial
join	pk_join

Relationships

Relationship	Role(s)
Nodes	nodes
Target	target
Source	source
PortUsages	portsUsed

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.4.5 «Success»

Inheritance

Behavioral_Elements::State_Machines::FinalState
«Success»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Nodes	nodes
Target	target
Source	source
PortUsages	portsUsed

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.4.6 «Failure»

Inheritance

Behavioral_Elements::State_Machines::FinalState
«Failure»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	**Role(s)**
Nodes	nodes
Target	target
Source	source
PortUsages	portsUsed

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

N/A

Diagram Notation

N/A

2.4.4.7 «enumeration» Status

Instantiation in a model

Concrete

Semantics

Corresponds to the enumeration of same name in the metamodel.

Enumeration Literals

Corresponding to the enumeration literals of the enumeration of same name in the metamodel,

- Success
- BusinessFailure
- TimeoutFailure
- TechnicalFailure
- AnyFailure
- AnyStatus

2.4.5 Stereotypes for Composition

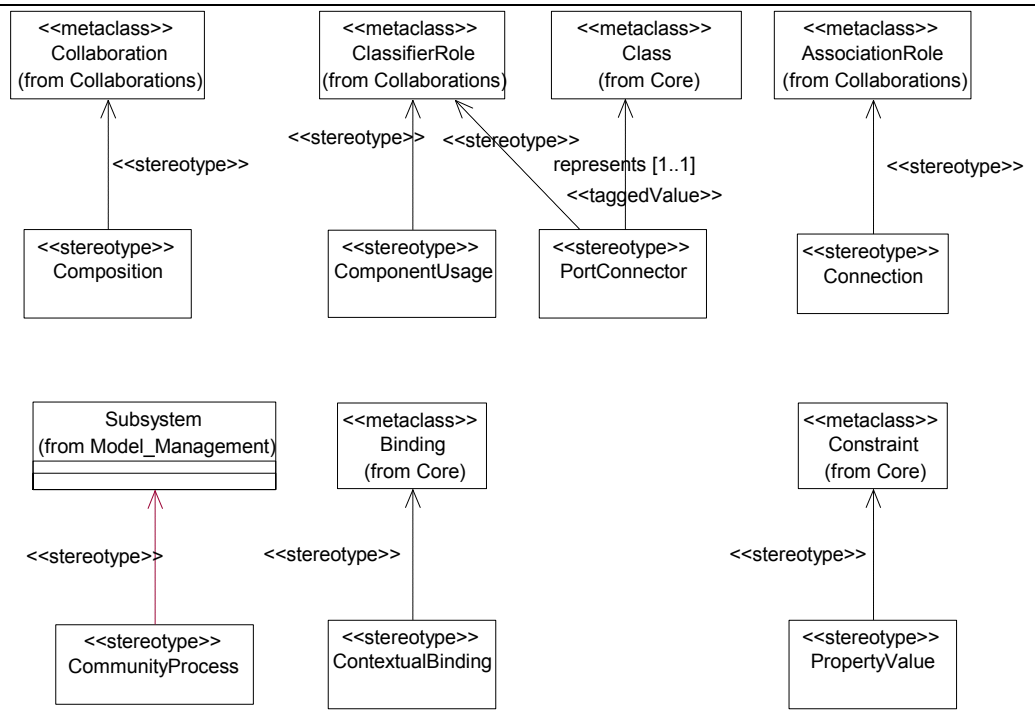

Figure 22: Stereotypes for Composition

Applicable Subset

Collaboration, ClassifierRole, AssociationRole, Constraint, Binding.

2.4.5.1 «Composition»

Inheritance

Behavioral_Elements::Collaborations::Collaboration
«Composition»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	**Role(s)**
Is_a_Composition	is_generalization
Generalization	parent child {only with

	«Composition»}
ComponentIUsages	owner
Nodes	_nodes
Connections	_connections
Bindings	owner
PackageElements	owner ownerElements
UML Namespace owner of «PortConnector»	ClassifierRoles

Tagged Values

N/A

Constraints expressed generically

The supertype of a «Composition» must be a «Composition».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context Composition

inv:
  supertype->isEmpty() or
  supertype.isStereoKinded("Composition")
```

Diagram Notation

N/A

2.4.5.2 «ComponentUsage»

Inheritance

Behavioral_Elements::Collaborations::ClassifierRole

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	**Role(s)**
Nodes	nodes
ComponentUsages	uses
Fills	fills

PortUsages	extent
Configuration	owner

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

```
context ComponentUsage
```

Diagram Notation

N/A

2.4.5.3 «PortConnector»

Inheritance

Behavioral_Elements::Collaborations::ClassifierRole
«PortConnector»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
PortUsages	PortsUsed, extent
Represents	_represents
Target	target
Source	source
Nodes	nodes

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
name	name	ModelElement	

Tagged Values

N/A

Constraints expressed generically

If the «Port» used by the «PortConnector» is a «FlowPort», and the «FlowPort» specifies a "typeProperty" (a «PropertyDefinition» in the owner «ProcessComponent»), then the actual type of the «PortConnector» will be a DataType, Enumeration, «CompositeData» or «ExternalDocument», with the name equal to the value of the «PropertyValue» of the «ComponentUsage» corresponding to the «PropertyDefinition» in the used «ProcessComponent».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context PortConnector
```

Diagram Notation

N/A

2.4.5.4 «Connection»

Inheritance

Behavioral_Elements::Collaborations::AssociationRole
«Connection»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of named "Connection" in the metamodel.

If one of the «Connection»s link participants is a «PortConnector» that "uses" a UML Classifier (corresponding to a metamodel Interface), then the UML Operation that will be invoked on the Classifier, is identified by a UML Message of a UML Interaction in the «Composition». The UML Message will have an action attribute initialized with a CallAction on the UML Operation.

Relationships

Relationship	Role(s)
Connections	connections
Source	outgoing
Target	incoming

Tagged Values

N/A

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

```
context Connection
```

Diagram Notation

N/A

2.4.5.5 «PropertyValue»

Inheritance

Foundation::Core::Constraint

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Configuration	configuration
ValueFor	_fills

Tagged Values

N/A

Constraints expressed generically

If the «PropertyValue» configures the value of a «PropertyDefinition» that is the "typeProperty" of a «FlowPort», then the value configured by the «PropertyValue» must be the name of a DataType, Enumeration, «CompositeData» or «ExternalDocument».

A «PropertyValue» is an ownedElement of a «Composition» as Namespace.

Formal Constraints Expressed in Terms of the UML Metamodel

```
context PropertyValue

  inv:
     namespace->notEmpty() and
  namespace.isStereoKinded("Composition")
```

Diagram Notation

N/A

2.4.5.6 «ContextualBinding»

Inheritance

Foundation::Core::Binding
«ContextualBinding»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

A «ContextualBinding» is an ownedElement of a «Composition».

The "client" of a ContextualBinding is a «ComponentUsage» in the «Composition».

The "supplier" of a ContextualBinding is a «ProcessComponent».

In the «Composition», the «ProcessComponent» will be used as the "uses" for the «ComponenUsage».

Relationships

N/A

Tagged Values

N/A

Constraints expressed generically

Formal Constraints Expressed in Terms of the UML Metamodel

```
context ContextualBinding
```

Diagram Notation

N/A

2.4.5.7 «CommunityProcess»

Inheritance

ModelManagement::Subsystem
«CommunityProcess»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

N/A

Tagged Values

N/A

Constraints expressed generically

Formal Constraints Expressed in Terms of the UML Metamodel

```
context CommunityProcess
```

Diagram Notation

N/A

2.4.6 DocumentModel «profile» Package

The metamodel elements named Attribute, DataType and Enumeration correspond to the UML model elements of the same name and are not stereotyped.

The metaattribute named "initialValue" of the metamodel Attribute, corresponds to the attribute of same name of UML Attribute.

The metaattribute named "required" and "many" of the metamodel Attribute, are combined as a UML Multiplicity. The MultiplicityRange, will have the "lower" attribute value equal to 0, if the corresponding metamodel Attribute has the "required" meta-attribute equal to false, and greater than 0, if "required" is true. The MultiplicityRange will have the "upper" attribute value equal to 1, if the corresponding metamodel Attribute has the "many" meta-attribute equal to false, and and greater than 1, if "many" is true.

The metamodel element named Enumeration has a metaattribute named "initial" and type EnumerationValue. In the UML Profile, the responsibility of specifying an initial value, is delegated to the UML Attribute with type equal to the Enumeration. The initialValue attribute, of type Expression, in UML Attribute will be used to specify the default initial value of Enumeration.

The metamodel element named Enumeration Value corresponds to the UML model element named EnumerationLiteral.

The metamodel Attribute and UML Attribute correspond to each other completely, with the exception of the meta-attribute named "isByValue".

To represent "isByValue", a TaggedDefinition of same name and type Boolean is defined, to be applied on UML Attribute.

The TaggedDefinition is defined without creating a Stereotype of Attribute.

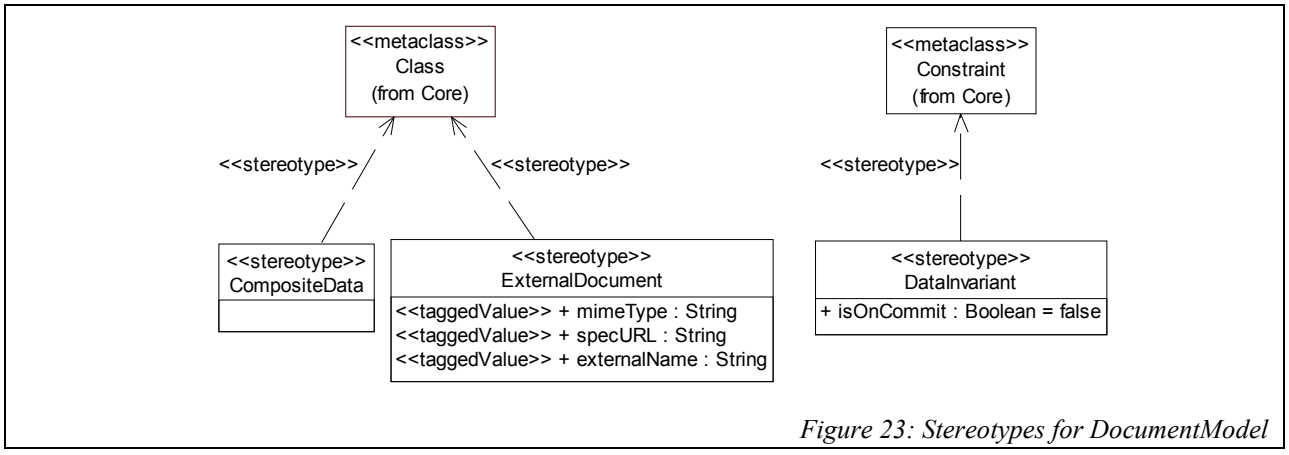

Figure 23: Stereotypes for DocumentModel

2.4.6.1 «CompositeData»

Inheritance

Foundation::Core::Class
«CompositeData»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

The «isByValue» TaggedDefinition can be applied to UML Attribute feature of «CompositeData».

Relationships

Relationship	Role(s)
Generalization	supertype subtypes {only with «CompositeData»}
PropertyType	type
AttributeType	type
DataAttribute	owner
DataConstraint	constrainedElement
FlowType	type
PackageContent	ownedElements
ImportElement	importedElement

Tagged Values

N/A

Constraints expressed generically

The supertype of an «CompositeData» must be a «CompositeData».

The type of Attributes of «CompositeData» will be a DataType, an Enumeration, or a Class stereotyped as «CompositeData», or a DataType stereotyped «ExternalDocument».

Formal Constraints Expressed in Terms of the UML Metamodel

```
context CompositeData

inv:
  supertype->isEmpty() or
  supertype.isStereoKinded("CompositeData")

inv:
```

```
feature->select( aFeature : Feature |
aFeature.isOCLTypeOf( Attribute))
->collect( aFeature : Feature | aFeature.oclAsType(
Attribute).type)
->forAll( aClassifier : Classifier |
   aClassifier.isOclKindOf( DataType) or
   aClassifier.isOclKindOf( Enumeration) or
   aClassifier.isStereoKinded( "CompositeData") or
   aClassifier.isStereoKinded( "ExternalDocument"))
```

Diagram Notation

N/A

2.4.6.2 "isByValue" Tagged Definition

The metamodel Attributes and UML Attributes correspond to each other completely, with the exception of the meta-attribute named "isByValue".

To represent the metamodel attribute named "isByValue", a Tagged Definition of named "isByValue" and type Boolean is defined, to be applied on UML Attribute.

The Tagged Definition is defined without creating a Stereotype of Attribute.

Tagged Value name	**Type**	**Multiplicity**	**Description**
isByValue	Boolean	0..1	default = true

2.4.6.3 «DataInvariant»

Inheritance

Foundation::Core::Constraint
«DataInvariant»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
DataConstraint	constrains

Correspondence of metamodel attributes with UML attributes

Metamodel attribute name	UML attribute name	UML attribute owner	Description
expression	body	Constraint	

Tagged Values

Tagged Value name	Type	Multiplicity	Description
isOnCommit	Boolean	1	default=false

Constraints expressed generically

N/A

Formal Constraints Expressed in Terms of the UML Metamodel

```
context DataInvariant
```

Diagram Notation

N/A

2.4.6.4 «ExternalDocument»

Inheritance

Foundation::Core::DataType
«ExternalDocument»

Instantiation in a model

Concrete

Semantics

Corresponds to the element of same name in the metamodel.

Relationships

Relationship	Role(s)
Generalization	supertype subtypes {only with «ExternalDocument»}
PropertyType	type
AttributeType	type
DataAttribute	owner
DataConstraint	constrainedElement
FlowType	type
PackageContent	ownedElements
ImportElement	importedElement

Tagged Values

Tagged Value name	Type	Multiplicity	Description
mimeType	String	0..1	
specURL	String	0..1	
externalName	String	0..1	

Constraints expressed generically

N/AFormal Constraints Expressed in Terms of the UML Metamodel

```
context ExternalDocument
```

Diagram Notation

N/A

2.4.7 UML Model_Management Package

There is no «profile» Package in the UML Profile for CCA, corresponding to the ModelManagement Package of the metamodel.

All the concrete metamodel elements have counterparts in UML, and therefore no stereotypes are required.

The metamodel elements named Package and ElementImport correspond to the UML model elements of the same name.

2.4.8 Relationships

This section specifies the correspondence between associations defined in the CCA Meta-model and associations defined in the UML Meta-model. The relationship name is the same as that found in the CCA Model diagrams (detail level). This correspondence is shown in the tables below, with a header for each relationship in the metamodel. This section provides detailed information for those implementing transformations between UML and MOF CCA tools, it is not required to use or understand CCA.

How to use this section.

Each relationship between two concepts in the metamodel, or their specializations, is represented with a UML relationship(s), and in some cases as a taggedValue, or by relating through UML Association.

The tables show the Left Hand and Right Hand sides of relationships, with the role names, the actual model elements at the ends of the relationship, and the specializations or stereotypes of interest, related through the relationship - directly or by inheritance. Multiple related metamodel elements or stereotypes may appear, at any side of relationships used by multiple elements.

The semantics of each row and column in the table are

- For each relationship in the metamodel, there is one or more tables, each table showing a particular mapping for that relationship. Each table has two lines – one for the CCA model (MOF) and one for the UML model (UML)
- For each relationship mapping in the metamodel :
- there is one row, labeled MOF, that describes the relationship in the metamodel. Its columns mean :
    - "LeftHandSide" in MOF rows, it names the MOF metamodel element that participates or inherits the relationship whose UML mapping we want to express. It may be the same as "LeftHandSide related", or a subtype of it. There may be multiple names, for various subtypes of polymorphically related metamodel elements.
    - "LeftHandSide related": in MOF rows, it names the actual metamodel element referenced by the relationship. May be the same as "LeftHandSide", or a supertype of it.
    - "LeftHandSide role name": in MOF rows, it names the relationship role on the LeftHandSide.
    - "RightHandSide role name": in MOF rows, it names the relationship role on the RightHandSide.
    - "RightHandSide related": in MOF rows it names the other actual MOF metamodel element referenced by the relationship. May be the same as 'RightHandSide", or a supertype of it.

- "RightHandSide": in MOF rows, it names the other metamodel element that participates or inherits the relationship whose UML mapping we want to express. It may be the same as in "RightHandSide related", or a subtype of it. There may be multiple names, for various subtypes of polymorphically related metamodel elements.

- row labeled 'UML' defining the corresponding UML Meta-model relationship. There may be additional tables for various UML mappings, describing alternative representations of the metamodel relationship in UML. The UML columns mean :
  - "LeftHandSide": In UML rows, it names the UML stereotype corresponding to the LHS MOF metamodel element. There may be multiple names, for various stereotypes and specializations.
  - "LeftHandSide related": In UML rows, it names the baseClass of the LHS UML stereotype, or the supertype of the baseClass, that is the actual UML model element referenced by the relationship.
  - "LeftHandSide role name": in UML rows, it names the relationship role on the LeftHandSide
  - "RightHandSide role name": in UML rows, it names the relationship role on the RightHandSide '.
  - "RightHandSide related": In UML rows, it names the baseClass of the RHS UML stereotype, or the supertype of the baseClass, that is the actual UML model element referenced by the relationship.
  - "RightHandSide": In UML rows, it names the UML stereotype corresponding to the RHS MOF metamodel element. There may be multiple names, for various stereotypes and specializations.

2.4.8.1 AttributeType

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Attribute	Attribute	_type	type	DataElement	DataType or Enumeration or CompositeData ExternalDocument
UML	«PropertyDefintion»	Attribute	typedFeature	type	Classifier	DataType or Enumeration or «CompositeData» «ExternalDocument»

2.4.8.2 Bindings

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Composition	Composition	owner	bindings	ContextualBinding	ContextualBinding
UML	«Composition »	Namespace	namespace	ownedElement	ModelElement	«ContextualBinding »

2.4.8.3 BindsTo

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ContextualBinding	ProcessComponent	bindsTo	bindsTo	ProcessComponent	ProcessComponent
UML	«ContextualBinding »	ModelElement	supplierDependency	supplier	ModelElement	«ProcessComponent »

2.4.8.4 Configuration

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ComponentUsage	ComponentUsage	owner	configuration	PropertyValue	PropertyValue
UML	«ComponentUsage»	ModelElement	constrainedElement	constraint	Constraint	«PropertyValue»

2.4.8.5 Connections in Choreography

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Choreography	Choreography	_choreography	connections	AbstractTransition	Transition
UML	«Choreography»	StateMachine or	stateMachine	transitions	Transition	Transition

2.4.8.6 Connections in Composition

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Composition	Choreography	_choreography	_connections	AbstractTransition	Transition
UML	«Composition»	Collaboration	namespace	ownedElement	AssociationRole	«Connection»

2.4.8.7 DataAtribute

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	CompositeData	CompositeData	owner	feature	DataElement	Attribute
UML	«CompositeData»	Classifier	owner	feature	Feature	Attribute

2.4.8.8 DataConstraint

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	DataInvariant	DataInvariant	constraints	constrainedElement	DataElement	DataElement subtypes: DataType or Enumeration or CompositeData or ExternalDocument

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
UML	«DataInvariant»	Constraint	constraint	constrainedElement	ModelElement	DataType or Enumeration or «CompositeData» or «ExternalDocument»

2.4.8.9 DataGeneralization

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	CompositeData	CompositeData	supertype	subtypes	CompositeData	CompositeData
UML	«CompositeData»	GeneralizableElement	generalization.parent	specialization. child	GeneralizableElement	«CompositeData»

2.4.8.10 Fills

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ContextualBinding	ProcessComponent	_fills	fills	ProcessComponent	ProcessComponent
UML	«ContextualBinding»	ModelElement	clientDeendency	fills	ModelElement	«ProcessComponent»

2.4.8.11 FlowType

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	FlowPort	FlowPort	_ type	type	DataElement	DataType or Enumeration or CompositeData or ExternalDocument
UML	«FlowPort»	ClassifierRole (indirectly thru AssociationEnd and Association)	association. association. connection. participant	association. association. connection. participant	ClassifierRole (indirectly thru AssociationEnd and Association)	DataType or Enumeration or «CompositeData» or «ExternalDocument»

2.4.8.12 Generalization

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent	Choreography	supertype	subtypes	Choreography	ProcessComponent
UML	«ProcessComponent»	GeneralizableElement	generalization.parent	specialization. child	Generalizable Element	«ProcessComponent»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Protocol	Choreography	supertype	subtypes	Choreography	Protocol
UML	«Protocol»	GeneralizableElement	generalization.parent	specialization. child	Generalizable Element	«Protocol»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	CommunityProcess	Choreography	supertype	subtypes	Choreography	CommunityProcess
UML	«CommunityProcess»	GeneralizableElement	generalization.parent	specialization. child	Generalizable Element	«CommunityProcess»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Interface	Choreography	supertype	subtypes	Choreography	Interface
UML	Classifier	GeneralizableElement	generalization.parent	specialization. child	Generalizable Element	Classifier

2.4.8.13 ImportElement

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ElementImport	ElementImport	elementImport	modelElement	PackageContent	Package or DataType or Enumeration or CompositeData or ExternalDocument or Protocol or Interface or ProcessComponent or CommunityProcess

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
UML	ElementImport	ElementImport	elementImport	importedElement	ModelElement	Package or DataType or Enumeration or «CompositeData» or «Protocol» or Classifier or «ProcessComponent» or «CommunityProcess»

2.4.8.14 Initiator

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Protocol or Interface	Protocol	_initiator	initiator	InitiatingRole	InitiatingRole
UML	«Protocol» or Classifier	Classifier	association. association. connection. participant	association. association. connection. participant	Classifier	«InitiatingRole»

2.4.8.15 Is_a_Choreography

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent or Protocol or Interface	ProcessComponent	is specialization	is generalization	Choreography	Choreography
UML	«ProcessComponent» or «Protocol» or Classifier	ModelElement	context	behavior	StateMachine	«Choreography»

2.4.8.16 Is_a_Composition

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent ComunityProcess	ProcessComponent	is specialization	is generalization	Composition	Composition
UML	«ProcessComponent» «ComunityProcess»	Classifier	represented Classifier	collaboration	Collaboration	«Composition»

2.4.8.17 Nodes in Choreograpy

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Choreography	Choreography	_choreography	_nodes	Node	PortActivity or Pseudostate
UML	«Choreography»	StateMachine	container. stateMachine container. container. ... stateMachine	top.subvertex top.subvertex. subvertex…	StateVertex	«PortActivity» or «Success» or «Failure» or Pseudostate

2.4.8.18 Nodes in Composition

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Choreography	Choreography	_choreography	_nodes	Node	PortActivity or Pseudostate

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
UML	«Choreography»	Composition	namespace	ownedElement	ClassifierRole	«PortActivity» or «Success» or «Failure» or Pseudostate

2.4.8.19 PackageElements

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Package ProcessComponent Protocol Interface CommunityProcess	Package	owner	ownedElements	PackageContent	Package or DataType or Enumeration or CompositeData or ExternalDocument or Protocol or Interface or ProcessComponent or CommunityProcess
UML	Package «ProcessComponent» «Protocol» Classifier «CommunityProcess»	Namespace	owner	ownedElement	ModelElement	Package or DataType or Enumeration or «CompositeData» or «Protocol» or Classifier or «ProcessComponent» or «CommunityProcess» indirectly through behavior.top.subvertex

2.4.8.20 Ports

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent or Protocol or MultiPort *	PortOwner	owner	ports	Port	FlowPort or ProtocolPort or MultiPort
UML	«ProcessComponent» or «Protocol» or «MultiPort»*	Classifier (indirectly thru AssociationEnd and Association)	association. association. connection. participant the Association may be stereotyped as «initiates» or «responds»	association. association. connection. participant	Classifier (indirectly thru AssociationEnd and Association)	«FlowPort» or «ProtocolPort» or «MultiPort»

(*) Constrained to «FlowPort». See Stereotype definitions, in sections above.

Additional Notes:

The MOF row is the description of the relationship in the metamodel:

> The ProcessComponent, Protocol and MultiPort inherits from PortOwner, and therefore has a role 'owner' in a relationship with Port, which participates in the relationship with the role name 'ports'. Specific subtypes of Port are FlowPort, ProtocolPort, OperationPort and MultiPort, that are related with ProcessComponent through the relationship inherited from Port.

The UML row identifies the UML relationships to represent the relationship in the metamodel, above.

> The stereotypes «ProcessComponent», «Protocol» and «MultiPort», corresponding to the metamodel elements of the same name, has a baseClass inheriting from Classifier, and therefore may be the participant in an AssociationEnd of a UML Association, with Classifier as the participant of the other AssociationEnd. The stereotypes with baseClass subtype of Classifier, «Port», «FlowPort», «ProtocolPort», and «MultiPort», corresponding to the metamodel elements of same name, are related with «ProcessComponent» through the said relationships with UML AssociationEnd and UML Association. MultiPort may only aggregate FlowPort.

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent or Protocol or Interface	PortOwner	owner	ports	Port	OperationPort
UML	«ProcessComponent» or «Protocol» or Classifier	Classifier	owner	feature	Feature	Operation

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	OperationPort	PortOwner	owner	ports	Port	Exactly one FlowPort with direction ="Responds"
UML	Operation	BehavioralFeature	behavioralFeature	parameter	Parameter	For each attribute of the «FlowPort».type a Parameter with kind=pdk_in and Parameter.type = the type of the Attribute

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	OperationPort	PortOwner	owner	ports	Port	At most one FlowPort with direction="Responds" and postCondition="Success"
UML	Operation	BehavioralFeature	behavioralFeature	parameter	Parameter	Parameter with Parameter.type= FlowPort.type and kind=pdk_return

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	OperationPort	PortOwner	owner	ports	Port with direction="Responds" and postCondition<>"Success"	FlowPort
UML	Operation	BehavioralFeature	context	raisedSignal	Signal	Signal with feature = «FlowPort».type.feature

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Interface	PortOwner	owner	ports	Port	OperationPort
UML	Classifier	Classifier	owner	feature	Feature	Operation

A metamodel Interface, owner of OperationPort, owner of FlowPort, map in the UML Profile, to a UML Classifier, owner of UML Operation, with UML Parameter with the type corresponding to the type of the metamodel FlowPort.

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	OperationPort	PortOwner	owner	ports	Port	FlowPort
UML	Operation	BehavioralFeature	behavioralFeature	parameter	Parameter	Parameter

2.4.8.21 PortUsages in Choreography

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent or Protocol	UsageContext	extent	portsUsed	PortUsage	PortActivity or Pseudostate
UML	«ProcessComponent» or «Protocol» indirectly through «Choreography»	ModelElement indirectly through StateMachine	indirectly through container. stateMachine. context	indirectly through behavior.top.subvertex	StateVertex indirectly through StateMachine	«PortActivity» or Pseudostate or «Success» or «Failure» indirectly through «Choreography»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PortActivity	UsageContext	extent	portsUsed	PortUsage	PortActivity or Pseudostate
UML	«PortActivity»	CompositeState	container	subvertex	StateVertex	«PortActivity» or Pseudostate or «Success» or «Failure»

2.4.8.22 PortUsages in Composition

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent	UsageContext	extent	portsUsed	PortUsage	PortConnector
UML	«ProcessComponent» indirectly through «Composition»	Classifier indirectly through Collaboration	indirectly through _representedClassifier. ownedElements	indirectly through owner. representedClassifier or owner.owner	ClassifierRole indirectly through Collaboration	«PortConnector» indirectly through «Composition»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ComponentUsage	UsageContext	extent	portsUsed	PortUsage	PortConnector
UML	«ComponentUsage»	ClassifierRole (indirectly thru AssociationEndRole and AssociationRole)	association. association. connection. participant	association. association. connection. participant	ClassifierRole (indirectly thru AssociationEndRole and AssociationRole)	«PortConnector»

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PortConnector	UsageContext	extent	portsUsed	PortUsage	PortConnector
UML	«PortConnector»	ClassifierRole (indirectly thru AssociationEndRole and AssociationRole)	association. association. connection. participant	association. association. connection. participant	ClassifierRole (indirectly thru AssociationEndRole and AssociationRole)	«PortConnector»

2.4.8.23 Properties

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProcessComponent	ProcessComponent	component	properties	PropertyDefinition	PropertyDefinition
UML	«ProcessComponent»	Classifier	owner	feature	StructuralFeature Attribute	«Property Definition»

2.4.8.24 PropertyType

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PropertyDefinition	PropertyDefinition	_type	type	DataElement	DataType or Enumeration or CompositeData ExternalDocument
UML	«PropertyDefintion»	Attribute	typedFeature	type	Classifier	DataType or Enumeration or «CompositeData» «ExternalDocument»

2.4.8.25 ProtocolType

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	ProtocolPort	ProtocolPort	_uses	uses	Protocol	Protocol

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
UML	«ProtocolPort»	GeneralizableElement	specialization.child	generalization.parent	Generalizable Element	«Protocol»

2.4.8.26 Represents in Choreography

The metamodel element Choreography is represented by a UML StateMachine, where a PortActivity in the metamodel is mapped to a stereotype of CompositeState.

The Represents relationship in the metamodel, that links a PortActivity with a Port, corresponds in UML to a TaggedValue of the Stereotype «PortActivity».

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	FlowPort or ProtocolPort or OperationPort or MultiPort	Port	represents	_represents	PortUsage	PortActivity
UML	«FlowPort» or «ProtocolPort» or «OperationPort » or «MultiPort»	Class	taggedValue "uses"	N/A : tagged values not bidirectional	SimpleState or CompositeState or SubmachineState or StubState or ActionState or SubactivityState	«PortActivity»

2.4.8.27 Represents in Composition

The metamodel element Composition is represented by a UML Collaboration.

A PortConnector is mapped to a ClassifierRole.

The "Represents" relationship linking a PortActivity with a Port, is represented in UML as a the UML relationship between a ClassifierRole and its base Classifier.

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	FlowPort or ProtocolPort or OperationPort or MultiPort	Port	represents	_represents	PortUsage	PortConnector
UML	«FlowPort» or «ProtocolPort» or «OperationPort » or «MultiPort»	Classifier	base	_base	ClassifierRole	«PortConnector»

2.4.8.28 Responder

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Protocol or Interface	Protocol	_initiator	initiator	RespondingRole	RespondingRole
UML	«Protocol» or Classifier	Classifier	association. association. connection. participant	association. association. connection. participant	Classifier	«RespondingRole»

2.4.8.29 Source

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PortActivity or Pseudostate	Node	target	incoming	AbstractTransition	Transition
UML	«PortActivity» or «Success» or «Failure» or Pseudostate	StateVertex	target	incoming	Transition	Transition

2.4.8.30 Target

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PortActivity or Pseudostate	Node	source	outgoing	AbstractTransition	Transition
UML	«PortActivity» or «Success» or «Failure» or Pseudostate	StateVertex	source	outgoing	Transition	Transition

2.4.8.31 TypeProperty

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	FlowPort	FlowPort	_typeProperty	typeProperty	PropertyDefinition	PropertyDefinition

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
UML	«FlowPort»	Class	N/A : tagged values not bidirectional	taggedValue named "typeExp"	Attribute	«PropertyDefinition»

2.4.8.32 Uses

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	Composition	Composition	owner	uses	ComponentUsage	ComponentUsage
UML	«Composition»	Namespace	owner	ownedElement	ModelElement	«ComponentUsage»

2.4.8.33 ValueFor

MOF or UML	LeftHandSide	LeftHandSide related	LeftHandSide role name	RightHandSide role name	RightHandSide related	RightHandSide
MOF	PropertyValue	PropertyValue	elementImport	fills	PropertyDefinition	PropertyDefinition
UML	«PropertyValue»	Constraint	elementImport	constrainedElement	ModelElement	PropertyDefinition

2.4.9 General OCL Definition Constraints

These definition constrains have been incorporated from the OMG Document ad/2000-02-02, UML Profile for CORBA, Joint Revised Submission Version 1.0 by Data Access Corporation, DSTC, Genesis Development Corporation, Telelogic AB, UBS AG, Lucent Technologies, Inc. and Persistence Software.

```
context ModelElement

  def:
     let allStereotypes : Set( Stereotype) =
                 -- set with the Stereotype applied to the
                 -- ModelElement and all the stereotypes
                 -- inherited by that Stereotype
           self.stereotype->union(

     self.stereotype.generalization.parent.allStereotypes)

     let isStereoTyped( theStereotypeName : String ) :
  Boolean =
                 -- returns true if an Stereotype
                 -- with name equalto the argument as been
                 -- applied to the ModelElement
           self.stereotype.name = theStereotypeName

     let isStereoKinded( theStereotypeName : String ) :
  Boolean =
                 -- returns true if an Stereotype with its
                 -- name equal to the argument, or equal to
                 -- any of its inherited Stereotypes,
                 -- has been applied to the ModelElement,
           self.allStereotypes->exists( aStereotype :
  Stereotype |
                 aStereotype.name = theStereotypeName)
```

2.5 Diagramming CCA

CCA models may be diagramed using generic as well as CCA specific notations. The generic notations (as found in UML 1.4) are supported by a wide variety of tools which allow CCA concepts to be made part of the larger enterprise picture without specific tool support. When using generic notations the CCA profile stereotypes should be used. CCA aware design & implementation tools may provide the CCA specific notation in addition to or instead of the other forms of notation.

This section suggests a non-normative way to utilize generic UML diagrams and CCA notation to express CCA concepts. For the generic diagrams it does so using an “out of the box” UML tool – Rational Rose 2000e ®.

2.5.1 Types of Diagram

The diagrams used to express CCA concepts are as follows:

2.5.1.1 Class Diagrams for the Document Model

These are used to express the document model.

2.5.1.2 Class Diagrams for the Component Structure

These are used to define components & protocols, their ports and properties.

2.5.1.3 Collaboration Diagrams for Composition

These are used to express the composition of components within another component or community processes.

2.5.1.4 State or Activity Diagrams for Protocols & Process Components

These express the ordering constraints on ports within or between components.

2.5.1.5 CCA Notation for Process Component Structure & Composition

This expresses the component structure and composition in a more compact and intuitive form, thus replacing the class and collaboration diagrams. We will show how the CCA notation expresses the same concepts found in the generic diagrams.

2.5.2 The Buy/Sell Example

The techniques for diagramming CCA will be presented by example. We will utilize a simple buy/sell business process to illustrate the concepts. We will summarize the points in the specification from the perspective of using a diagramming tool.

The basic business problem of buy/sell is to define a “community process” with two actors – a buyer and seller. These two actors “collaborate” within this process to effect an order.

2.5.3 Collaboration diagram shows community process

At the highest level we show a collaboration diagram of the Buy/Sell community process. In the design tool we also created a package for this process to hold the relevant model elements. See Figure 24.

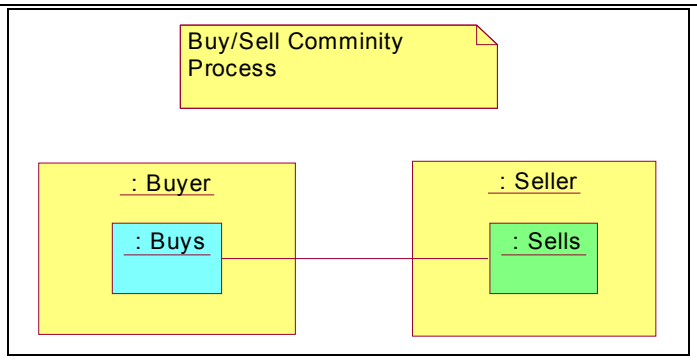

Figure 24: Top Level Collaboration Diagram

This collaboration shows both business roles: "Buyer" and "Seller". These are each a "ComponentUsage" in the CCA Meta-model. It also shown that the buyer has a "buys" port and the seller has a "sells" port that are connected by a Connection in this collaboration. The "buys" and "sells" ports are "PortConnectors" in the CCA Meta-model. The line between "Buys" and "sells" indicates that the buyer and seller collaborate on these ports using a "Connection".

There is no way to show which port is the initiator and which is the responder in a collaboration diagram, so we have noted the "buys" in blue and "sells" in green, for those of you who have color (for others you may be able to tell from the shade).

Note that "buys" and "sells" are shown inside of "buyer" and "seller", respectively. The use of this nested classifier notation shown that the ports are owned by the component. We could have also shown the ports separately with a connected line, but nesting them seems to better reflect the underlying semantics.

The design tool we are using does not show stereotypes in a collaboration diagram, if they did show you would see that buyer and seller have the <<ComponentUsage>> stereotype and "Buys" and "Sells" have the <<PortConnector>> stereotype. You would also see that the entire package has the stereotype <<CommunityProcess>>.

The following is a summary of the elements, stereotypes and base elements you would use in a collaboration diagram for a community process:

2.5.3.1 Summary of stereotypes for a Community Process

CCA element	Stereotype	Base UML Element	Example Elements
CommunityProcess	<<CommunityProcess>>	Package or Subsystem	BuySell
ComponentUsage	<<ComponentUsage>>	Classifier Role (Object*)	Buyer, Seller
PortConnector	<<PortConnector>>	Classifier Role (Object*)	Buys, Sells
Connection	None	Association Role (Object Link*)	Link from buys to sells
ContextualBinding	<<ContextualBinding>>	Binding (Note*)	None – used to refine which component type to use

CCA element	Stereotype	Base UML Element	Example Elements
PropertyValue	<<PropertyValue>>	Constraint (Note*)	None – use to set a configuration property of a component

Table 10: Summary of stereotypes for a Community Process

* *Denotes the name used in the design tool*

2.5.4 Class diagram for protocol structure

The buys and sells ports seen in the community process must have a prescribed protocol, a description of what information flows between them. This is shown in a class diagram (Figure 25). Additional information as to when information flows between them is shown on an associated state or activity diagram. The class diagram can include the definition of the data that flows between them (the document model), or this information can be shown on a separate class diagram.

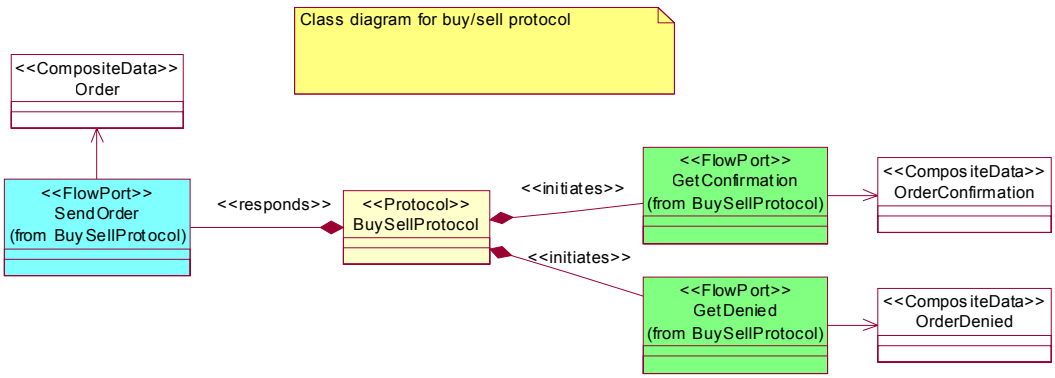

Figure 25: Class diagram for protocol structure

This diagram shows the protocol as well as the data used in the protocol (detail suppressed for this view). The protocol is a class stereotyped as <<Protocol>>. It has a set of flow ports: SendOrder, GetConfirmation, GetDenied. Each of these flow ports has an association to the data that flows over it; Order, OrderConfirmation and OrderDenied – respectivly.

A very important aspect of a port is its direction (initiates or responds), which is a tagged value. Since these tagged values don't sow on the diagram we have also stereotyped the relation to the ports as either <<initates>> or <<responds>> and have changed their color as was done in the collaboration diagram.

What this diagram shows is that implementers of the protocol "BuySellProtocol" will receive a "SendOrder" containing an "Order" and will send out a "GetConfirmation" (with data "OrderConfirmation") and/or a "GetDenied" (with data "OrderDenied").

The following is a summary of the elements, stereotypes and base elements you would use in a collaboration diagram for a protocol:

2.5.4.1 Summary of stereotypes for a Protocol

CCA element	Stereotype	Base UML Element	Example Elements
Protocol	<<Protocol>>	Class or Subsystem	BuySellProtocol
FlowPort	<<FlowPort>>	Class	SendOrder, GetConfirmation, GetDenied
"Ports" relation	Optional: <<initiates>> or <<responds>>	Association	Lines between FlowPorts and BuySellProtocol
ProtocolPort	<<ProtocolPort>>	Class	None – used to nest one protocol in another
OperationPort	<<OperationPort>>	Class	None – used to define a two-way message (could have been used for BuySell)
InitiatingRole	<<InitiatingRole>> with relation to protocol	Class	None – Used to name the initiating "side" of the protocol (the client)
RespondingRole	<<RespondingRole>> with relation to protocol	Class	None – Used to name the responding "side" of the protocol (the service)
Interface	Optional: <<Interface>>	Classifier	None – defines an object service
Direction (value)	<<initiatiates>>	Association	SendOrder
Direction (value)	<<responds>>	Association	OrderConfirmation, OrderDenied

Table 11: Summary of stereotypes for a Protocol

2.5.4.2 Summary of tagged values for a Protocol

While tagged values can't be seen in the diagram, these elements will have tagged values. The tagged values used to define a protocol are:

CCA attribute	Tagged Vale	Applies to	Example Values
synchronous	synchronous	FlowPort, ProtocolPort, OperationPort, MultiPort	All ports Synchronous=false (The response may come back at a later time)
transactional	transactional	FlowPort, ProtocolPort, OperationPort, MultiPort	True for all ports – each interaction is atomic.
direction	direction	FlowPort, ProtocolPort, OperationPort, MultiPort	Initiates for SendOrder. responds for GetConfirmation & GetDenied

CCA attribute	Tagged Vale	Applies to	Example Values
postCondition	postcondition	FlowPort, ProtocolPort, OperationPort, MultiPort	GetConfirmation=Success GetDenied=BusinessFailure

Table 12: Summary of tagged values for a Protocol

2.5.5 Activity Diagram (Choreography) for a Protocol

The class diagram for a protocol (Figure 26) shows what the protocol will send and receive but not when. The activity diagram of the prtocol adds this information by specifying when each port will perform its activity (sending and receiving information).

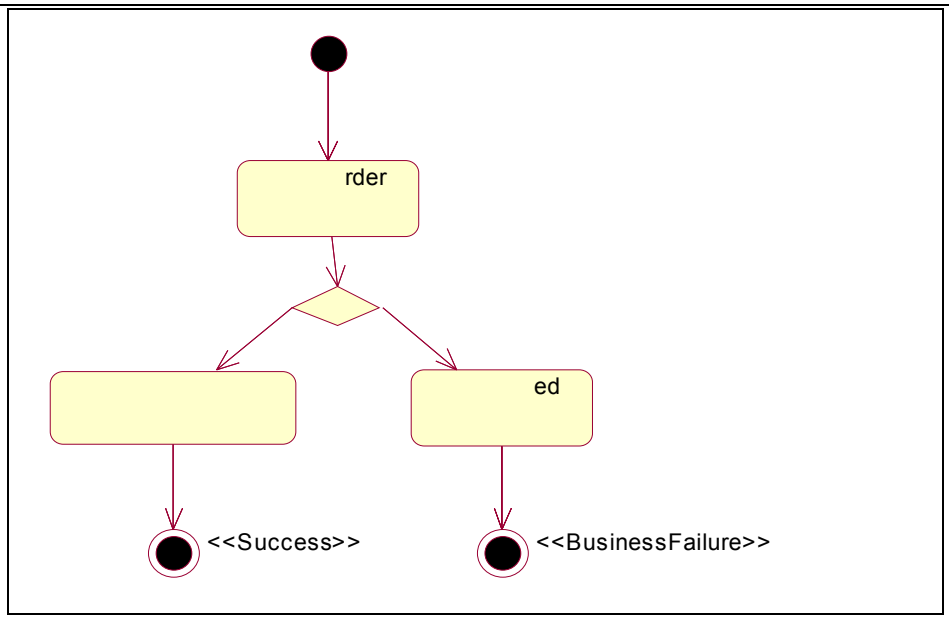

Figure 26: Choreography of a Protocol

As you can see, the activity diagram for the protocol is quite simple, it shows the start state, one activiation of each port and the transitions between them. It also shows that after the "SendOrder" a choice is made and either "GetConfirmation" or "GetDenied" is activated, but not both.

The start state (Black circle) shown where the protocol will start. It then goes to a "PortActivity" for the SnedOrder port (the port and the activity have the same name in this case). It then shows a choice (the diamond) and PortAcitivites for GetConfirmation and GetDenied ports. It then shows that either of these ends the protocol, but that GetConfirmation ends it with the status of Business Success while GetDenied ends it with BusinessFailure. (Success and failure can be tested in later transitions, using a guard on the transition). The transitions (each of the arrows) clearly shows the flow of control in the protocol.

Note that if there are multiple activities for one port it may be convenient to use swim lanes, one for each port. But swim lanes are not required.

What can not be seen is that each PortActivity has a tagged value: “represents” to connect it to the port it is an activity of. In the example “represents” will be the same as the activity name.

2.5.5.1 Summary of stereotypes for an Activity Diagram or Choreography

CCA element	Stereotype	Base UML Element	Example Elements
Choreography	<<Choreography>>	StateMachine	BuySellProtocol (not visible)
PortActivity	<<PortActivity>>	State	SendOrder, GetConfirmation, GetDenied
Psedostate (initial)	None (Black circle)	Psedostate (initial)	Start state
Psedostate (fork)	None (bar)	Psedostate (fork)	None – shows concurrency in process
Psedostate (join)	None (bar)	Psedostate (join)	None – shows concurrency coming together.
Psedostate (choice)	None (diamond)	Psedostate (choice)	Choice of confirm or denied.
Transition	<<ChoreographyTransition>>	Transition	All arrows

Table 13: Stereotypes for an Activity Diagram or Choreography

2.5.5.2 Summary of tagged values for a Choreography

While tagged values can’t be seen in the diagram, these elements will have tagged values. The tagged values used to define a Choreography are:

CCA attribute	Tagged Vale	Applies to	Example Values
represents	<<represents>>	PortActivity	All Activities Represents has the same value as element name

Table 14: Tagged Values for a Choreography

2.5.6 Class Diagram for Component Structure

The external “contract” of a component is shown on two diagrams – the class diagram for structure and the activity diagram for Choreography (much like the protocol). The structure shows the process component(s), their ports and properties.

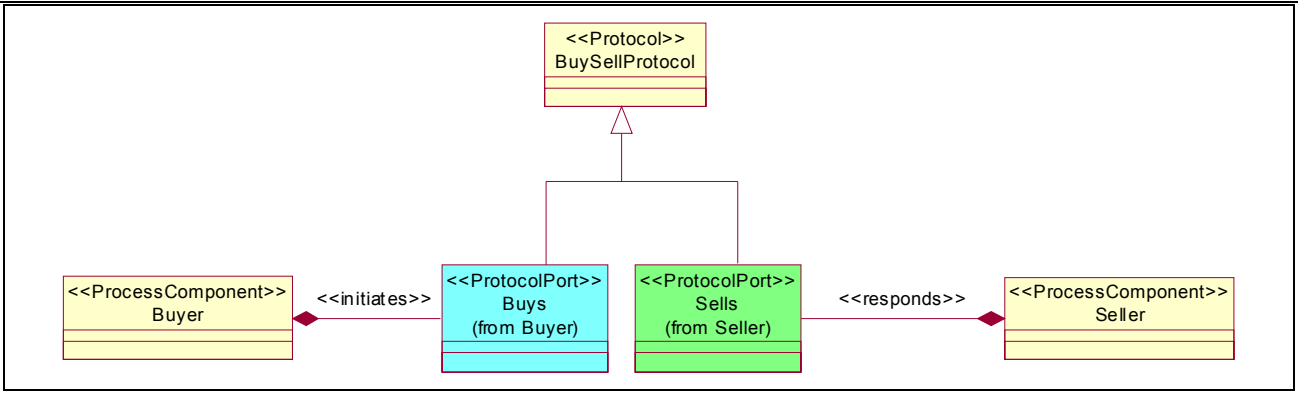

Figure 27: Class Diagram for Component Structure

This class diagram shows two process components being defined: "Buyer" and "Seller". Each process component uses the "ProcessComponent" stereotype. It also shows that each of these components has one protocol port each: "Buys" and "Sells", respectively and that both of these ProtocolPorts implement the BuySellProtocol we saw earlier.

We can also see that the buyer "initiates" the protocol via the "Buys" port and that the seller "responds" to (or implements) that interface via the "Sells" port. As before, both ports will have their direction set in a tagged value – the color and stereotypes on relations is just informational.

You may also note that we choose to define the ports as nested classes of their process components, as can be seen from the phrases (from Buyer) and (from Seller). This helps organize the classes but is purely optional.

These components are the ones we saw being used inside of the community process.

2.5.6.1 Summary of stereotypes for a Process Component Class Diagram

CCA element		Base UML Element	Example Elements
ProcessComponent	<<ProcessComponent>>	StateMachine	Buyer, Seller
FlowPort	<<FlowPort>>	Class	None – for primitive flows
"Ports" relation	Optional: <<initiates>> or <<responds>>	Association	Associations between ProtcolPorts and ProcessComponents
ProtocolPort	<<ProtocolPort>>	Class	Buys, Sells
OperationPort	<<OperationPort>>	Class	None – used to define a two-way message
MultiPort	<<MultiPort>>	Class	None – Shows a set of ports with a behavioral constraint
PropertyDefinition	<<PropertyDefiinition>>	Attribute	None – shows a configuration value
Direction (value)	<<initiatiates>>	Association	Buyer
Direction (value)	<<responds>>	Association	Seller

Table 15: Stereotypes for a Process Component Class Diagram

2.5.6.2 Summary of tagged values for a Process Component Class Diagram

While tagged values can't be seen in the diagram, these elements will have tagged values. The tagged values used to define a process component are:

CCA attribute	Tagged Vale	Applies to	Example Values
granularity	granularity	ProcessComponent	Buyer & Seller are "shared"
isPersistent	isPersistent	ProcessComponent	Buyer & Seller are persistent
primitiveKind	PrimitiveKind	ProcessComponent	Buyer & Seller are not primitive so have no primitiveKind.
primitiveSpec	PrimitiveSpec	ProcessComponent	Buyer & Seller are not primitive so have no primitiveSpec
synchronous	synchronous	FlowPort, ProtocolPort, OperationPort, MultiPort	All ports Synchronous=false (The response may come back at a later time)
transactional	transactional	FlowPort, ProtocolPort, OperationPort, MultiPort	True for all ports – each interaction is atomic.
direction	direction	FlowPort, ProtocolPort, OperationPort, MultiPort	Initiates for Buys responds for Sells
postCondition	postcondition	FlowPort, ProtocolPort, OperationPort, MultiPort	N/A
initial	None: UML "Initial Value"	PropertyDefinition	None
isLocked	None: UML changability	PropertyDefinition	None

Table 16: tagged values for a Process Component Class Diagram

2.5.7 Class Diagram for Interface

Classical "services" are provided for with the CCA "Interface", such a service interface corresponds to the normal concept of an object. An interface is a one-way version of a protocol and may not have sub-protocols. Once such service is defined for our example.

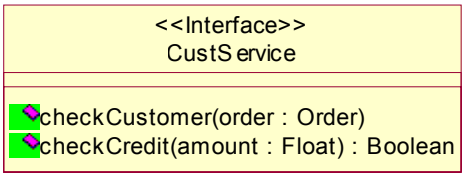

Figure 28: Class Diagram for Interface

Since the semantics of such an interface are will understood, let's just relate to the CCA elements:

Example Element		**UML Element**
CustService	Interface	Interface
CheckCustomer	FlowPort	Operation
CheckCustomer. order	DataElement	Parameter
checkCredit	OperationPort	Operation
CheckCredit. anount	FlowPort	Parameter

Table 17: Elements of an Interface

Note that the use of a stereotype for an interface is optional., allowing the use of other forms of UML classifiers.

Interfaces may have the same tagged values as protcol, but interfaces don't need "direction", the direction is always "responds".

2.5.7.1 Using Interfaces

While we are on the subject, let's also look at the class diagram for a process component with a port that implements this interface.

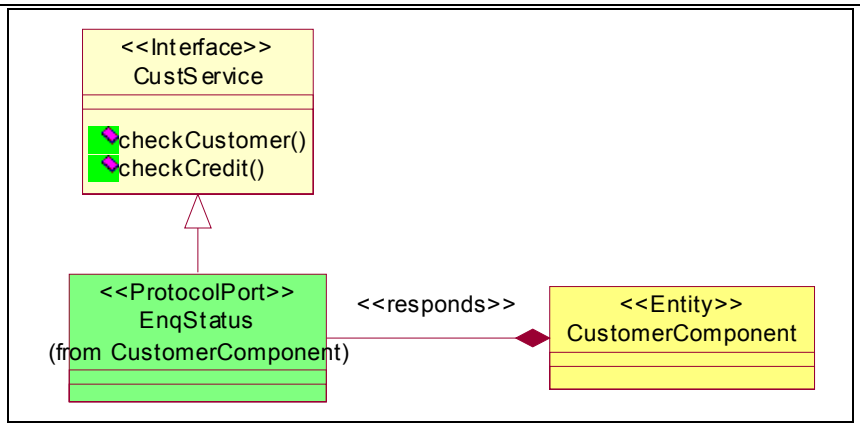

Figure 29: Using Interfaces

This diagram shown an "Entity" ProcessComponent (see entity profile) called "CustomerComponent" which exposes a ProtocolPort (EnqStatus) which implements this interface.

2.5.8 Class Diagram for Process Components with multiple ports

Up to this point we have seen process components with only one port, while most process components interact with multiple other components. We are going to define such a component that will be used inside other components later.

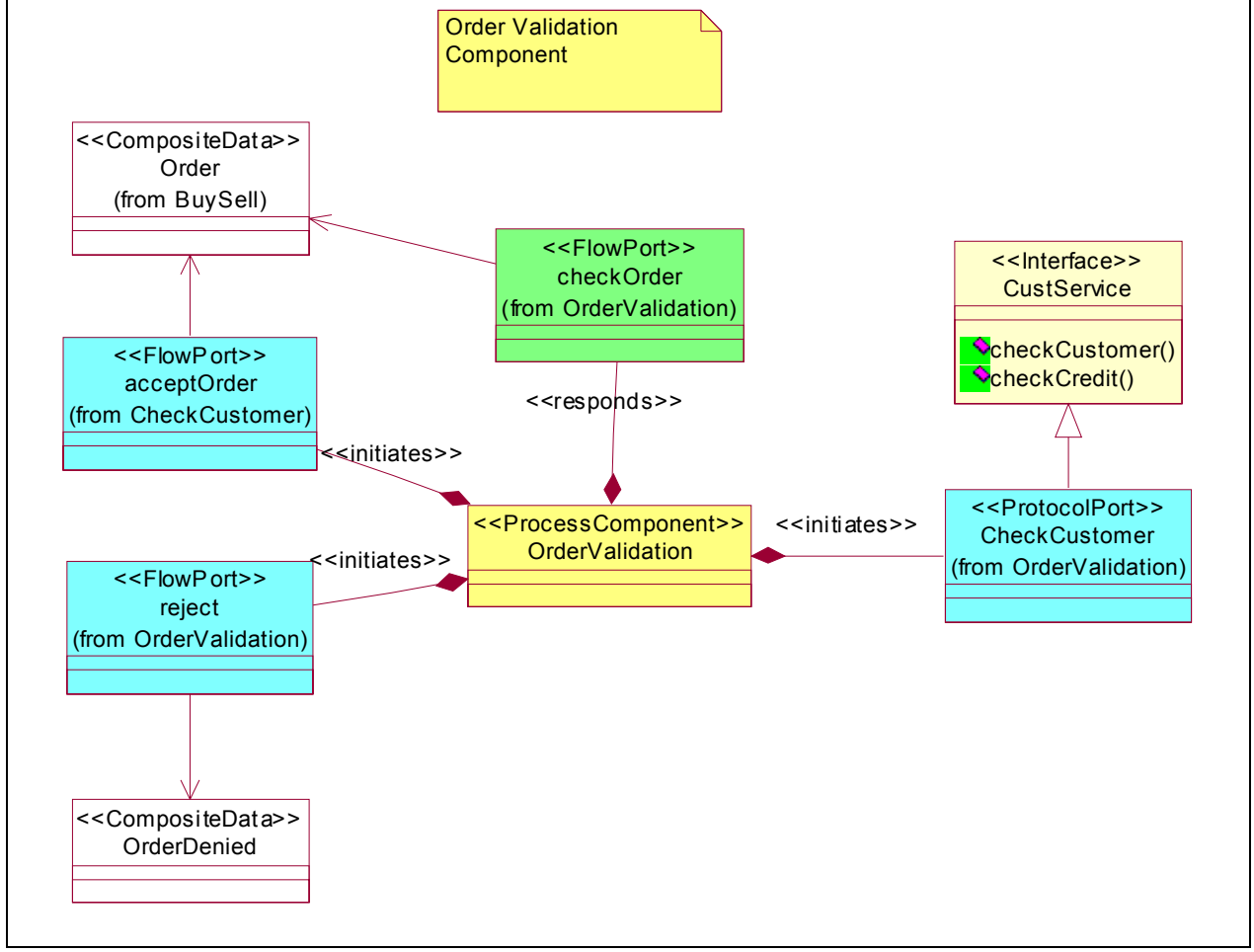

Figure 30: Process Components with multiple ports

This diagram defines the OrderValidation ProcessComponent. Note that it has the following ports:

- checkOrder – responding flow port (the order)
- CheckCustomer – initiating protocol port to a service
- AcceptOrder – intiating flow port (the order)
- Reject – initiating flow port (OrderDenied)

2.5.9 Activity Diagram showing the Choreography of a Process Component

Since our Order Validation process component has multiple ports, we may also want to specify the choreography of those ports, when each will activate. This is done using an activity diagram much like the protocol.

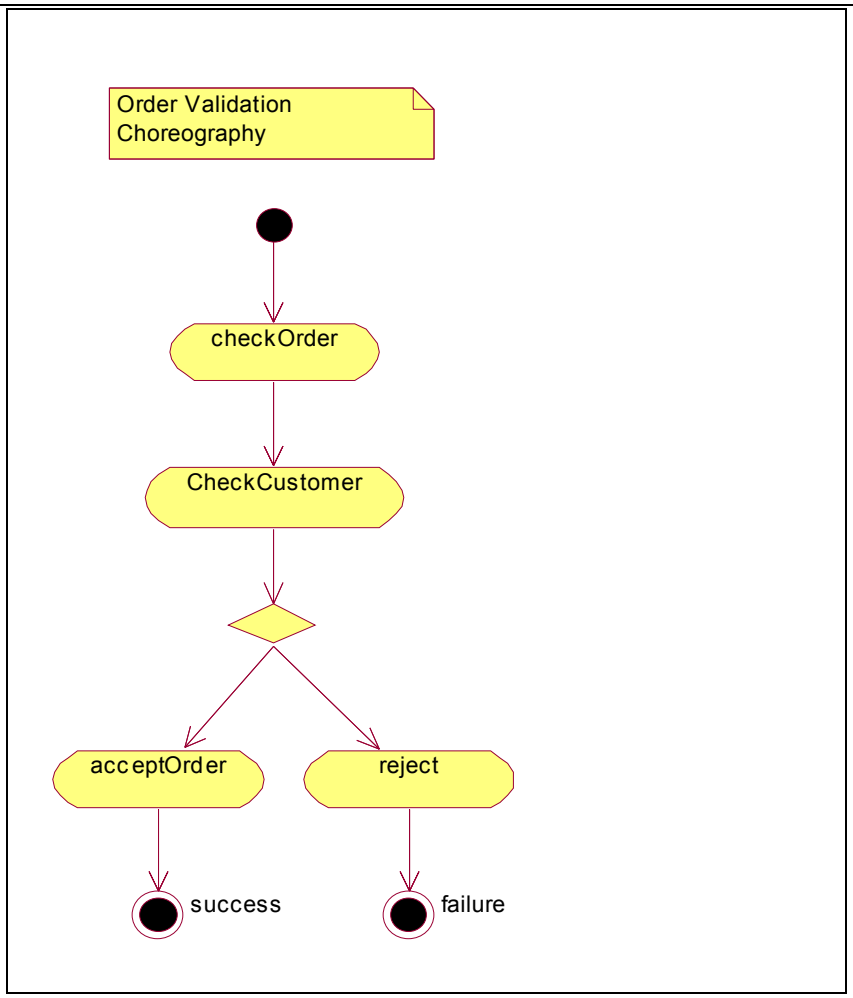

Figure 31: Choreography of a Process Component

Since the model elements used here are the same as those for the protocol, we will not repeat the tables.

2.5.10 Collaboration Diagram for Process Component Composition

A composition collaboration diagram shows how components are used to help define and (perhaps) implement another component. We have already seen one composition, for the community process. Now we will look at a collaboration diagram which specifies the inside of one of our process components – the seller.

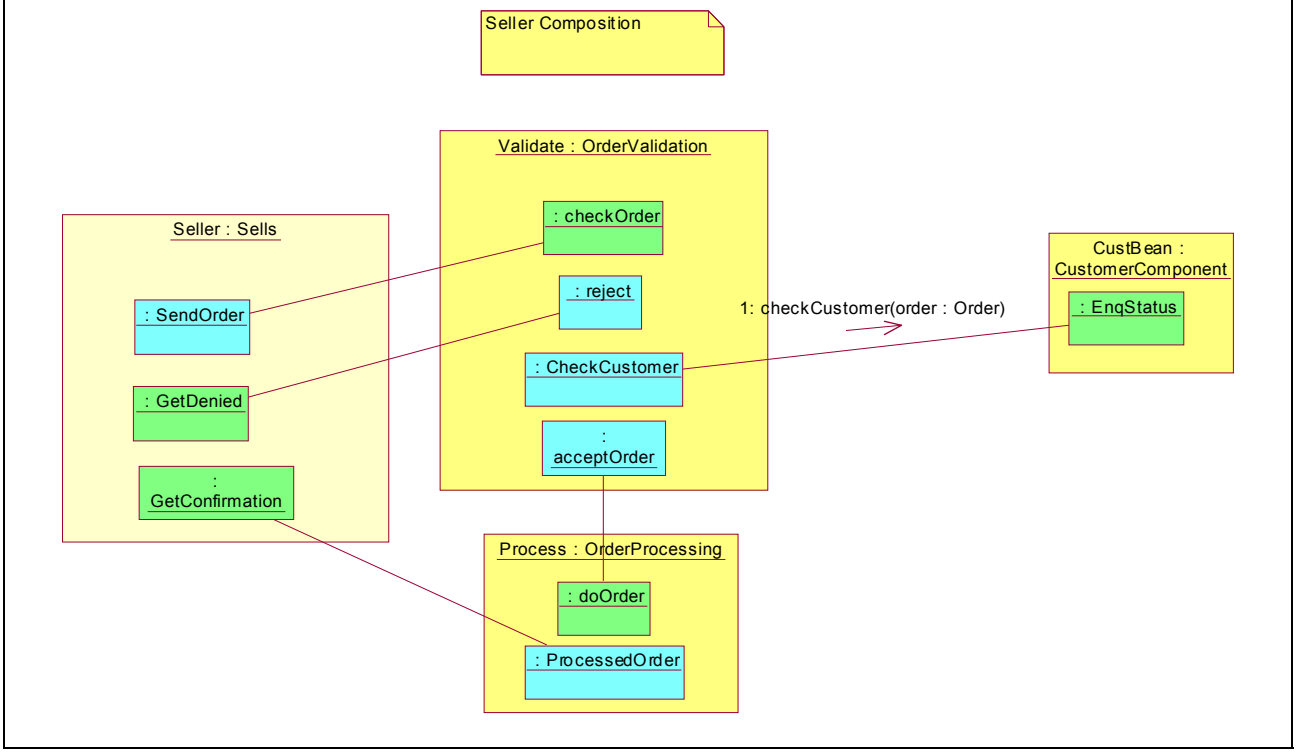

Figure 32: Process Component Composition

This is a collaboration diagram "inside" the seller, which the seller will do to implement its protocol by *using* other components. This is a very specific use of a collaboration diagram and needs some explanation.

First note that, like the community process, we are showing the ports of components and of protocols nested inside the component or protocol.

The Component Usages are as follows:

- Validate – uses the "OrderValidation" component
- CustBean – uses the CustomerComponent
- Process – uses the "OderProcessing" component (not previously shown)

If we look inside of "Validate" we see a classifier role for each port: checkOrder, reject, CheckCustomer & acceptOrder. We see the same pattern repeated inside of CustBean and Process.

Note "Seller : Sells". This is the representation of the "Sells" port on the component being defined – in this case "Seller". There will be such a "proxy" PortConnector for each port on the outside of the component for which we are making the collaboration diagram. Since this port is a protocol port, it also has sub-ports which show up as nested classifier roles.

To "connect" one port to another we draw an association role (a line representing a Connection) from one port to another. The connected ports must have compatible types and directions. So in this diagram we have made the following connections:

2.5.10.1 Connections in the example

From Component Usage	From Port Connector	To Port Connector	To Component Usage
Seller	Sells	CheckOrder	Validate
CheckOrder	Reject	GetDenied	Seller
Validate	CheckCustomer	EnqStatus * Using Operation "checkCust"	CustBean
Validate	AcceptOrder	DoOrder	Process
Process	ProcessOrder	GetConfirmation	Seller

Table 18: Connections

Each of these connections will cause data to flow from one component to the other, via the selected ports. It is these Connections which connect the activities of the components together in the context of this composition.

2.5.10.2 Summary of stereotypes for a Process Component Collaboration

CCA element	Stereotype	Base UML Element	Example Elements
Composition	<<Composition>>	Collaboration	Seller Composition
ProcessComponent	Implied	Classifier	Seller
ComponentUsage	<<ComponentUsage>>	Classifier Role (Object*)	Validate, Process, CustBean
PortConnector	<<PortConnector>>	Classifier Role (Object*)	Seller, SendOrder, GetDenied, GetConfirmation CheckOrder, reject, CheckCustomer, acceptOrder DoOrder, ProcessOrder EnqStatus
Connection	Connection (Optional)	Association Role (Object Link*)	See above table
ContextualBinding	<<ContextualBinding>>	Binding (Note*)	None – used to refine which component type to use
PropertyValue	<<PropertyValue>>	Constraint (Note*)	None – use to set a configuration property of a component

Table 19: Stereotypes for a Process Component Collaboration

2.5.10.3 Special note on "proxy" port activities.

As can be seen from the example, we need to connect the "outside" ports (those on the component being defined) with the "inside" ports (those on the components being used). The PortConnectors for the outside ports are shown without an owning ComponentUsage, while the PortConnectors for the components being used are shown inside of the ComponentUsage being used.

2.5.10.4 Special note on protocols

Since protocols give us the ability to "nest" ports, ports may be seen within ports to any level. This example only shown one level of such nesting. The same kind of nesting is used within activity diagrams – since activities may be nested as well.

2.5.11 Model Management

While the organizational structure of components is not visible in a diagram, it is visible in tools. The screen shot in Figure 33 shows how the example components are organized in the Data Access Technologies' UML tool. Note how using nested classes (such as Ports being inside of their ProcessComponent) helps to organize the model and keep namespaces separate.

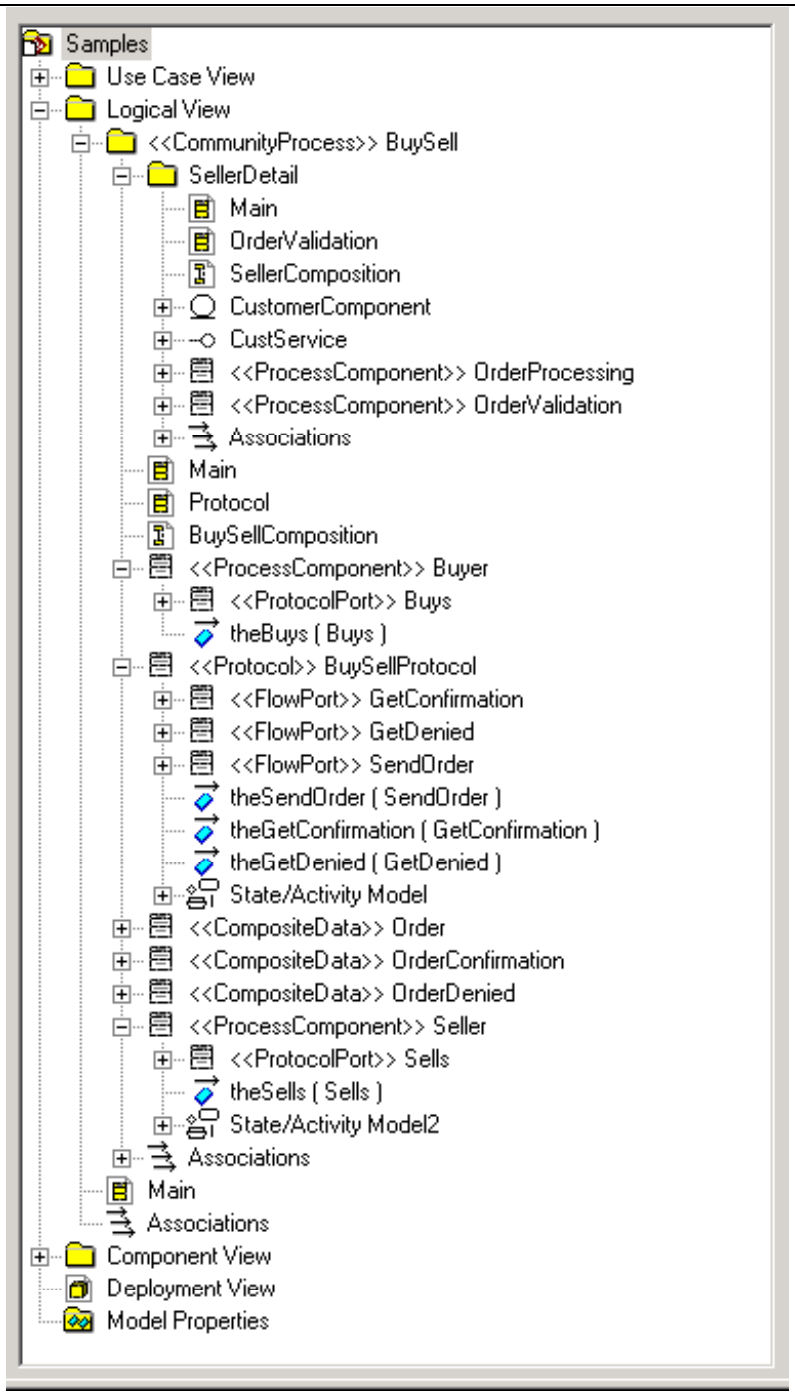

Figure 33: Model Management

2.5.12 Using the CCA Notation for Component & Protocol Structure

Figure 34 shows the CCA notation being used for the protocol and process component structure, above. Note that as with the UML notation, this is done from an out-of-the-box tool (Component-X®) - the notation is not quite standard CCA yet.

This shows the community process and protocol corresponding to the UML example, above.

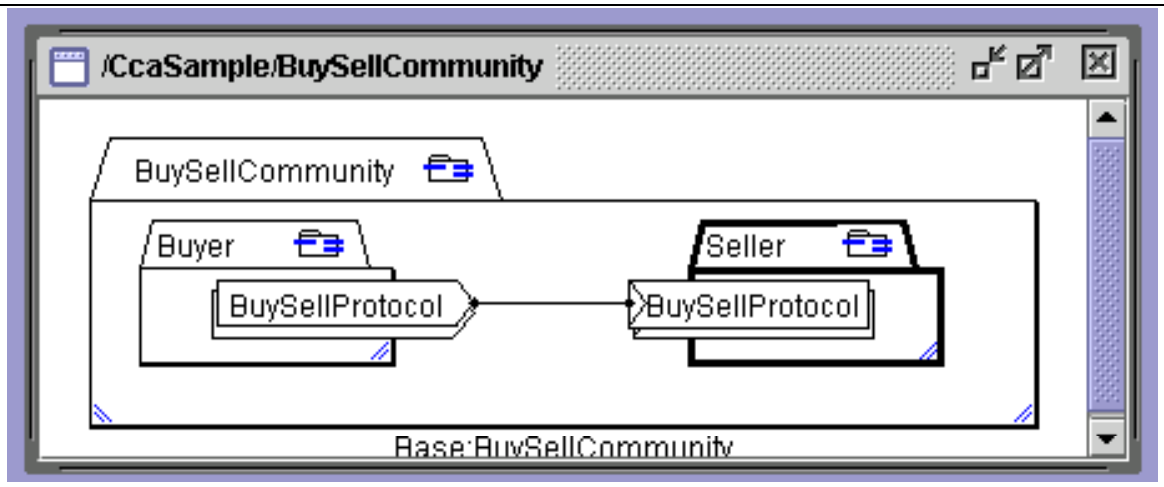

Figure 34: Community Process and Protocol

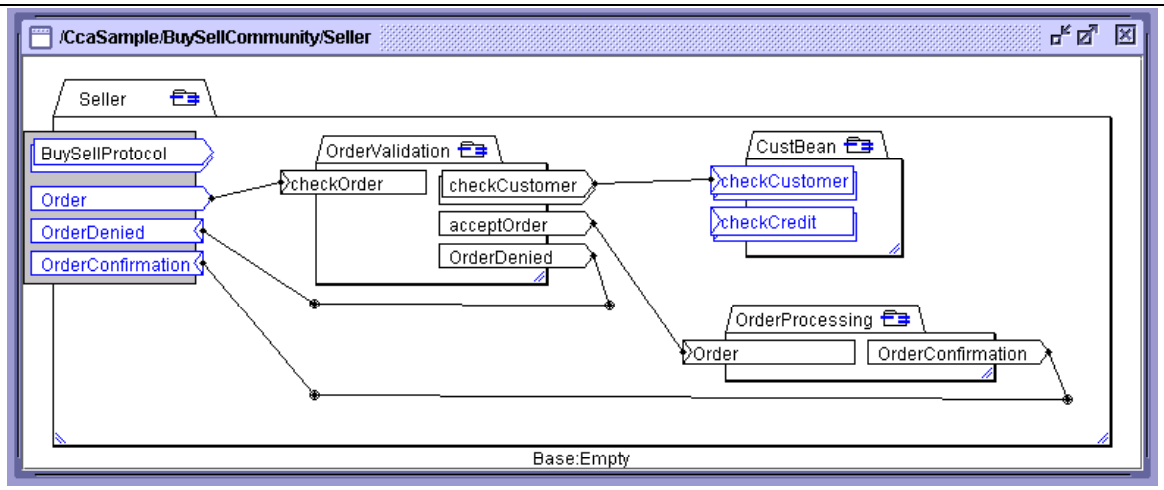

Figure 35 Composition in CCA notation

Figure 35 shows the seller composition in CCA notation; it is equivalent to the seller collaboration diagram.

3. The Sales example

This example illustrates the specification of a system of collaborating parties, involved in a commercial Sale.

The Sales example defines the collaboration between the parties involved.

The focus is on the boundaries between the parties – ComponentUsages, their specification – ProcessComponents, their connectable point – Ports, and the externally observable contract of candidate interactions – Protocols .

Each party may be further specified as an internal composition of collaborating sub-components, onto which the external contract is delegated.

3.1 Performer for the ProcessOrder Activity of the Procurement System example

The Sales example is referenced as part of the Procurement Process of the Buyer, as the Performer for the ProcessOrder Activity..

Please refer to the Procurement System example of the Business Processes Profile (Section 2 above), for the specification of the Business Process of the Buyer, where this Sales example is used and initiated, to fulfill the ProcessOrder Activity.

In the context of the Buyer Business Process :

(copied from the Procurement System example (Section 2))

"… After the Authorizing Officer has awarded the contract to a particular supplier, the order is released to that supplier for processing. …"

The organization performing the Procurement Process plays the role of Buyer, and the awarded supplier plays the role of Seller, in the BuySellCommunity CommunityProcess.

The Award Activity will determine the identity of the actual Seller instance, corresponding to a ProcessComponent type of Seller, that plays the Seller role in the BuySell CommunityProcess.

3.2 BuySell Community Process

The BuySell CommunityProcess specifies how a Buyer, a Seller and a Logistics collaborate to complete a business. Each role is played by a ComponentUsage of the same name. The specifications for the used ProcessComponent can be found under headers below.

The Buyer collaborates directly with the Seller, through the Buy and Sell ProtocolPorts, according to the Sales Protocol.

The Seller and the Buyer collaborate with the Logistics, through the Ship and Delivery ProtocolPorts, according to Protocol of the same names. The specification for the Protocols can be found under headers below.

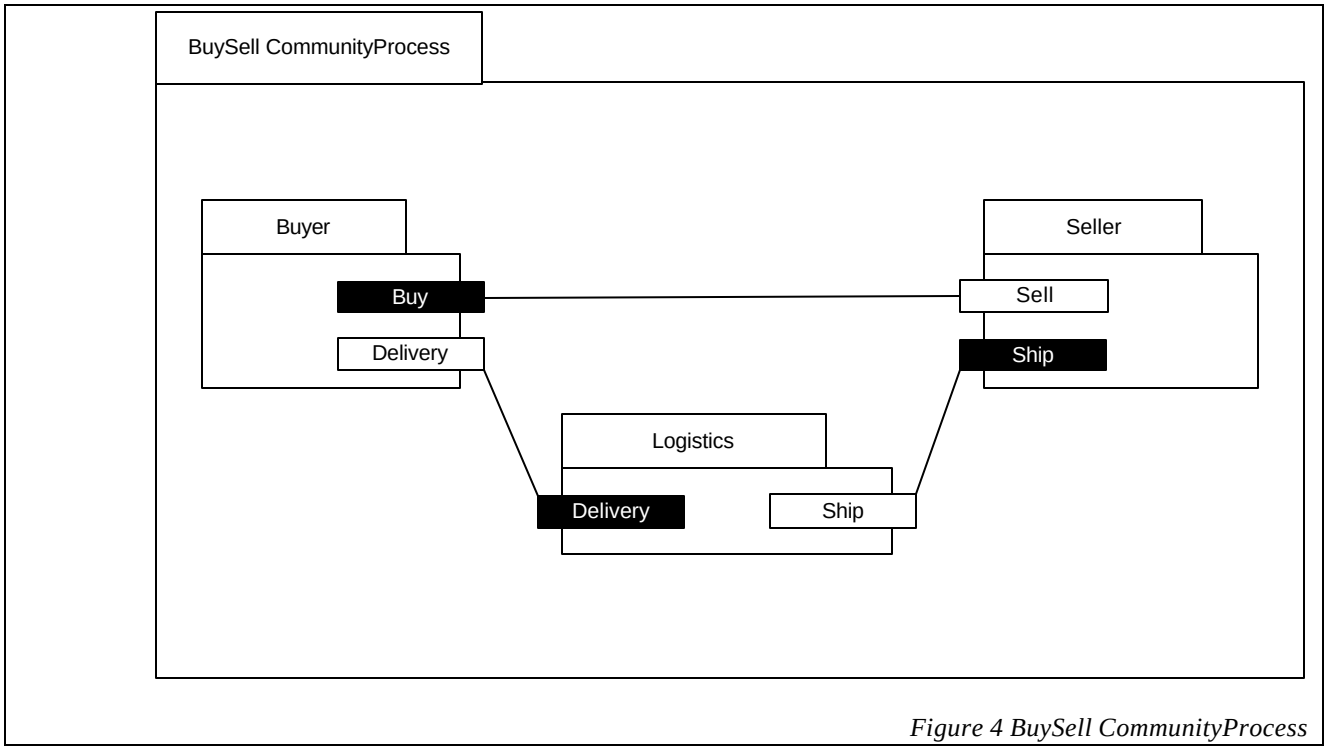

Figure 4 BuySell CommunityProcess

The activities in the BuySell Community Process start by the Buyer initiating the interactions on its Buy ProtocolPort, according to the Sales Protocol.

The Seller is connected through its Sell ProtocolPort, to the Buy ProtocolPort of the Buyer. Therefore, the Seller will respond to the Sales Protocol, as initiated from the Buyer.

The Seller will follow the Sales Protocol, and eventually initiate the Ship Protocol with the Logistics role. The Logistics role will respond to the Ship Protocol, and initiate the Delivery Protocol on the Buyer. The Buyer will then be able to proceed with the Sales Protocol, and complete the overall collaboration.

3.3 Protocols

3.3.1 Sales Protocol

The interactions between the ComponentUsage in the BuySell CommunityProcess, above, occur according to Protocols, as specified below.

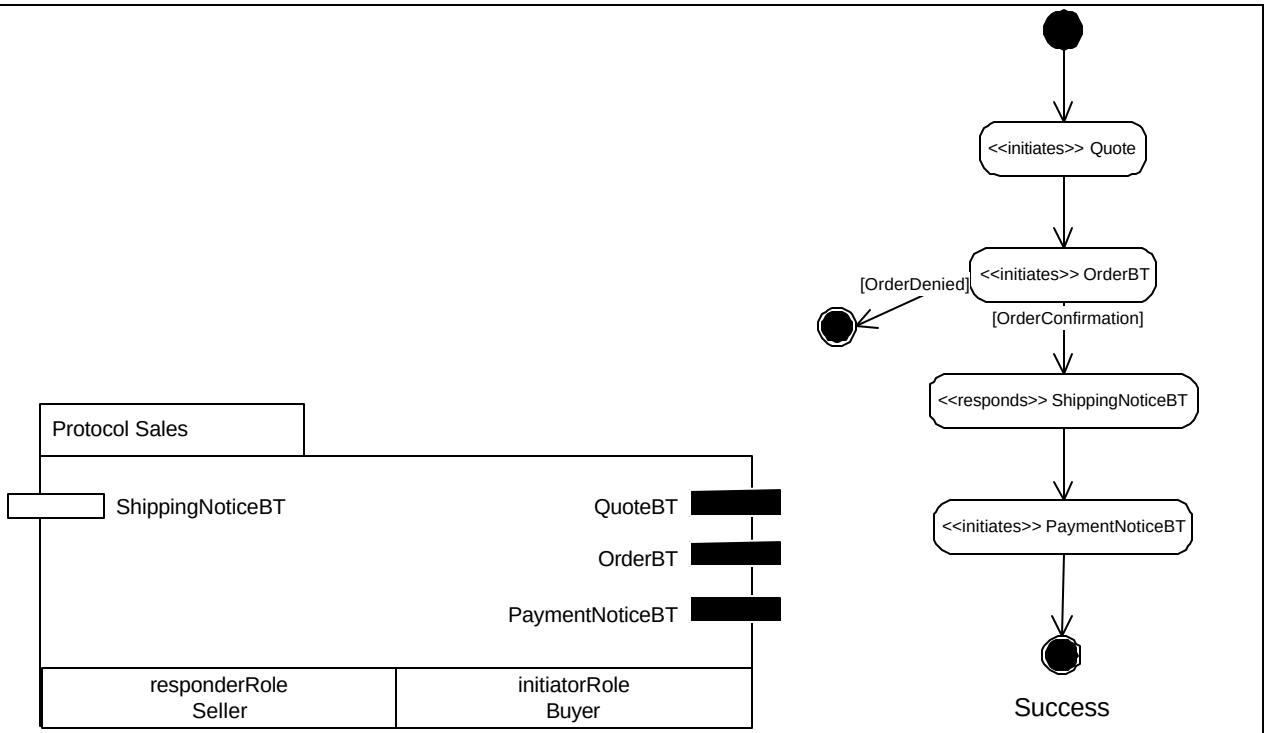

Figure 5 Sales Protocol structure and choreography

Structure

The Sales Protocol is an integration of four simpler Protocols : QuoteBT, OrderBT and PaymentNoticeBT. The Sales Protocol has a ProtocolPort using each of these simpler Protocols. The specification for these Protocols can be found under headers below.

Interactions in the ProtocolPorts QuoteBT, OrderBT and PaymentNoticeBT will be initiated by the initiatorRole of the Sales Protocol.

The initiatorRole of the Sales Protocol will respond to interactions in the ShippingNoticeBT ProtocolPort.

Choreography

Interactions in the Sales Protocol will begin by the initiatorRole of the Sales Protocol, initiating and fully performing the interactions of the QuoteBT ProtocolPort.

After this, the initiatorRole will initiate and fully perform the interactions of the OrderBT ProtocolPort.

If during performance of the interaction of the OrderBT ProtocolPort, an OrderDenied has flown between initiatorRole and responderRole, then the Protocol ends with a Failure condition.

Else, if an OrderConfirmation has flown, then the initiatorRole of the Sales Protocol will respond and fully perform the interactions of the ShippingNoticeBT ProtocolPort.

After this, the initiatorRole will initiate and fully perform the interactions in the PaymentNoticeBT ProtocolPort.

3.3.2 QuoteBT Protocol

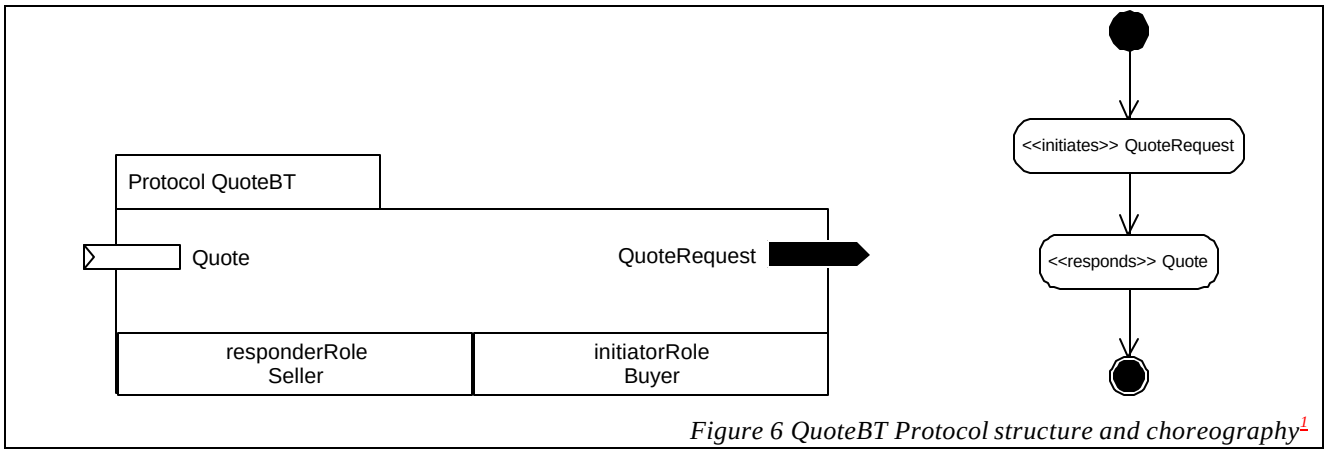

Figure 6 QuoteBT Protocol structure and choreography[1]

QuoteBT is a Protocol in the form of a Request-Reply, where the initiatorRole will send a QuoteRequest, and receive a Quote as response. QuoteRequest and Quote are FlowPort of the QuoteBT Protocol, typed to CompositeData of the same name.

3.3.3 OrderBT Protocol

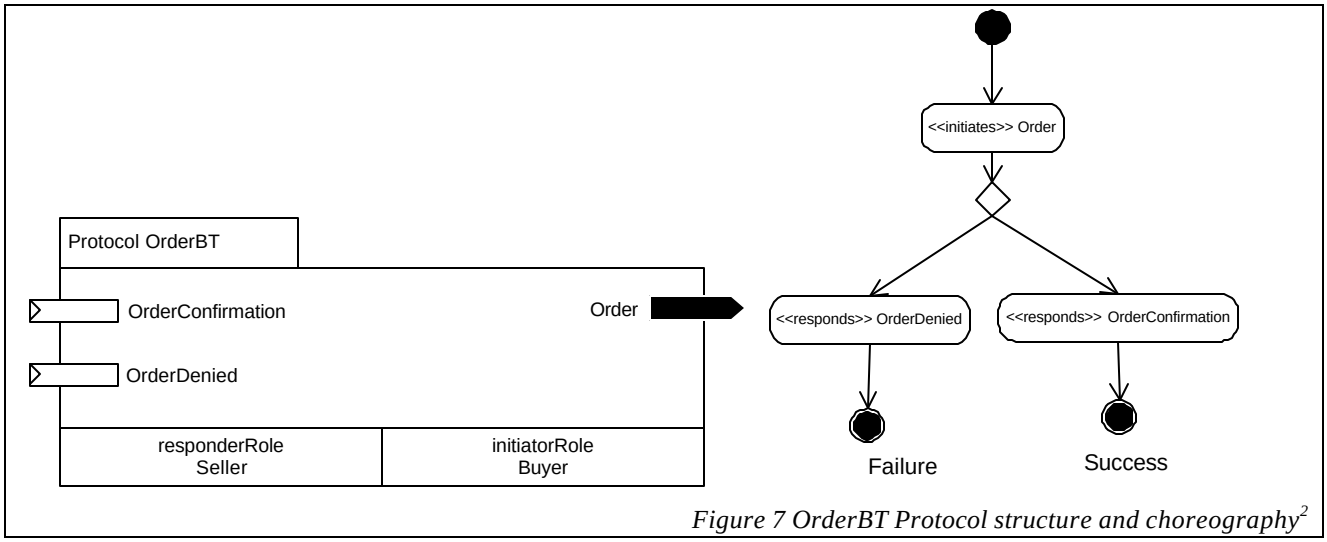

Figure 7 OrderBT Protocol structure and choreography[2]

QuoteBT is a Protocol in the form of a Request-Multiple_Candidate_Reply, where the initiatorRole will send an Order, and receive as response an OrderConfirmation or an OrderDenied. Order, OrderConfirmation and OrderDenied are FlowPort of the OrderBT Protocol, typed to CompositeData of the same name.

[1] The direction of the ports is incorrect in Figures 6 to 11. In all these diagrams, <<responds>> should read <<initiates>>, and *vice versa*.

[2] See footnote to Figure 6

An OrderConfirmation leads to a successful termination of the Protocol, while an OrderDenied is a Failure condition.

3.3.4 ShippingNoticeBT Protocol

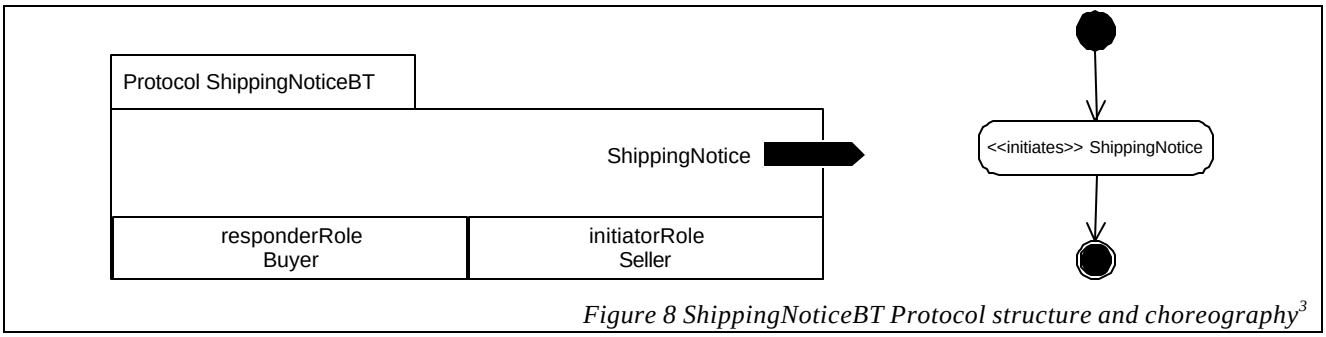

Figure 8 ShippingNoticeBT Protocol structure and choreography[3]

ShippingNoticeBT is a Protocol with a single FlowPort, corresponding to the sending of a ShippingNotice by the initiatorRole of the Protocol.

To declare a Protocol for a single flow may be redundant, as the unique FlowPort could be included wherever the Protocol is used, like in the Sales Protocol of our example. In this case, ShippingNoticeBT has been defined, for symmetry, and to illustrate the benefit of this approach, encapsulating as a Protocol the single flow nature of the interaction.

3.3.5 PaymentNoticeBT Protocol

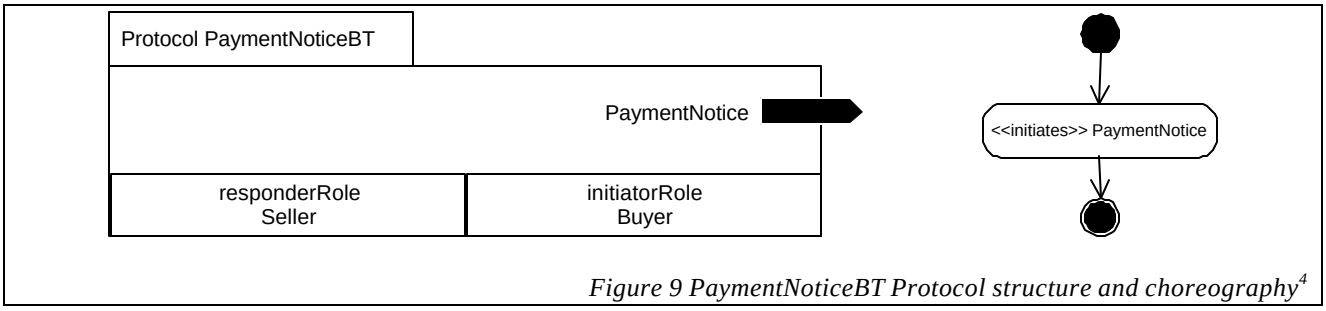

Figure 9 PaymentNoticeBT Protocol structure and choreography[4]

PaymentNoticeBT is a Protocol with a single FlowPort, corresponding to the sending of a PaymentNotice by the initiatorRole of the Protocol.

[3] See footnote to Figure 6
[4] See footnote to Figure 6

3.3.6 ShipBT Protocol

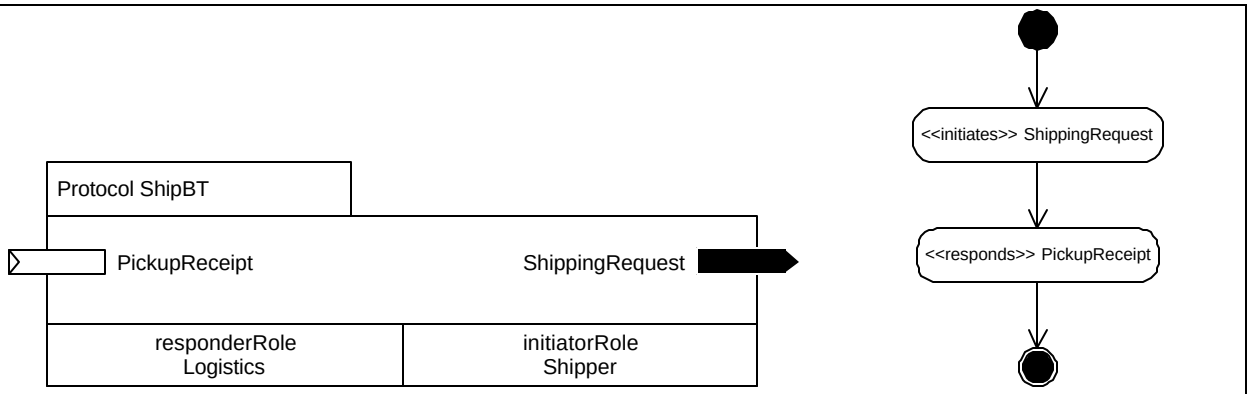

Figure 10 ShipBT Protoco structure and choreography[5]

ShipBT is a Protocol in the form of a Request-Reply, where the initiatorRole will send a ShippingRequest, and receive a PickupReceipt as response. ShippingRequest and PickupReceipt are FlowPort of the ShipBT Protocol, typed to CompositeData of the same name.

3.3.7 DeliveryBT Protocol

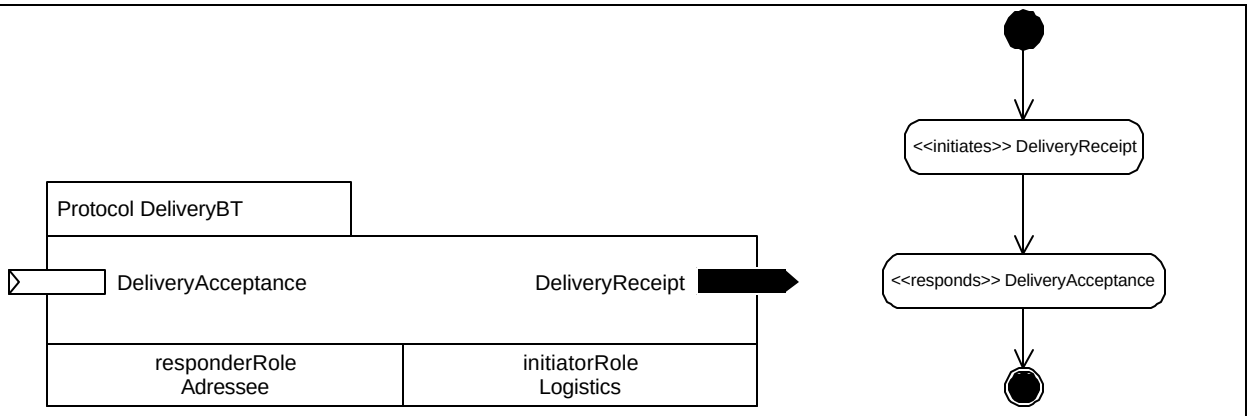

Figure 11 DeliveryBT Protocol structure and choreography[6]

DeliveryBT is a Protocol in the form of a Request-Reply, where the initiatorRole will send a DeliveryReceipt, and receive a DeliveryAcceptance as response. DeliveryReceipt and DeliveryAcceptance are FlowPort of the DeliveryBT Protocol, typed to CompositeData of the same name.

[5] See footnote to Figure 6
[6] See footnote to Figure 6

3.4 Components

3.4.1 Buyer ProcessComponent

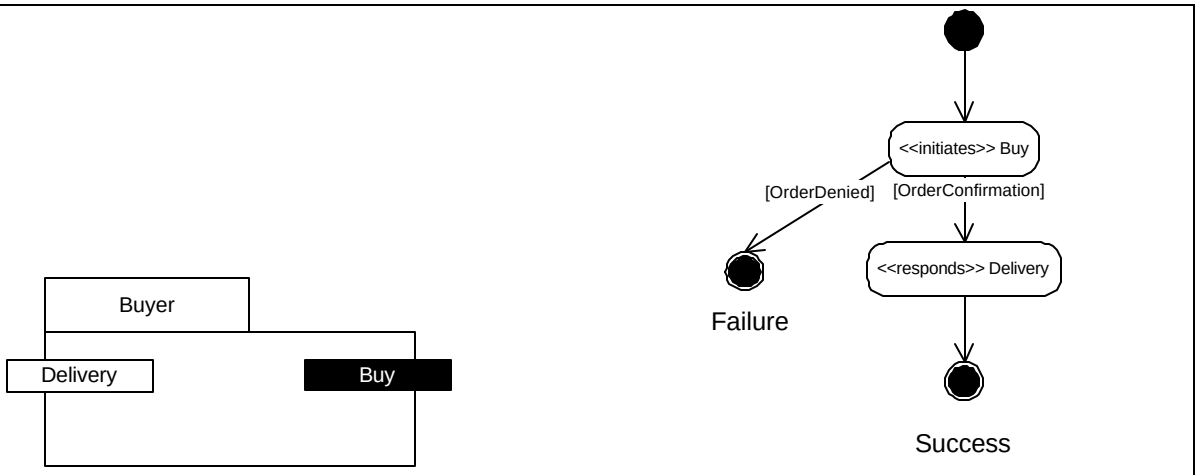

Figure 12 Buyer ProcessComponent structure and choreography

Buyer ProcessComponent is used in the BuySell CommunityProcess, as ComponentUsage of the same name.

Buyer has two ProtocolPort named Buy and Delivery.

The Buyer initiates interactions through the Buy ProtocolPort according to the Sales Protocol. The Delivery ProtocolPort responds to the DeliveryBT Protocol.

The activities of the Buyer ProcessComponent will begin by initiating and fully performing the interactions through the Buy Port, according to the used Sales Protocol.

After this, if during performance of the interaction of the Sales Protocol through the Buy ProtocolPort, an OrderDenied has flown, then the choreography ends with a Failure condition.

Else, if an OrderConfirmation has flown, then the Buyer ProcessComponent will respond to interactions through the Delivery ProtocolPort, and complete successfully.

3.4.2 Seller ProcessComponent

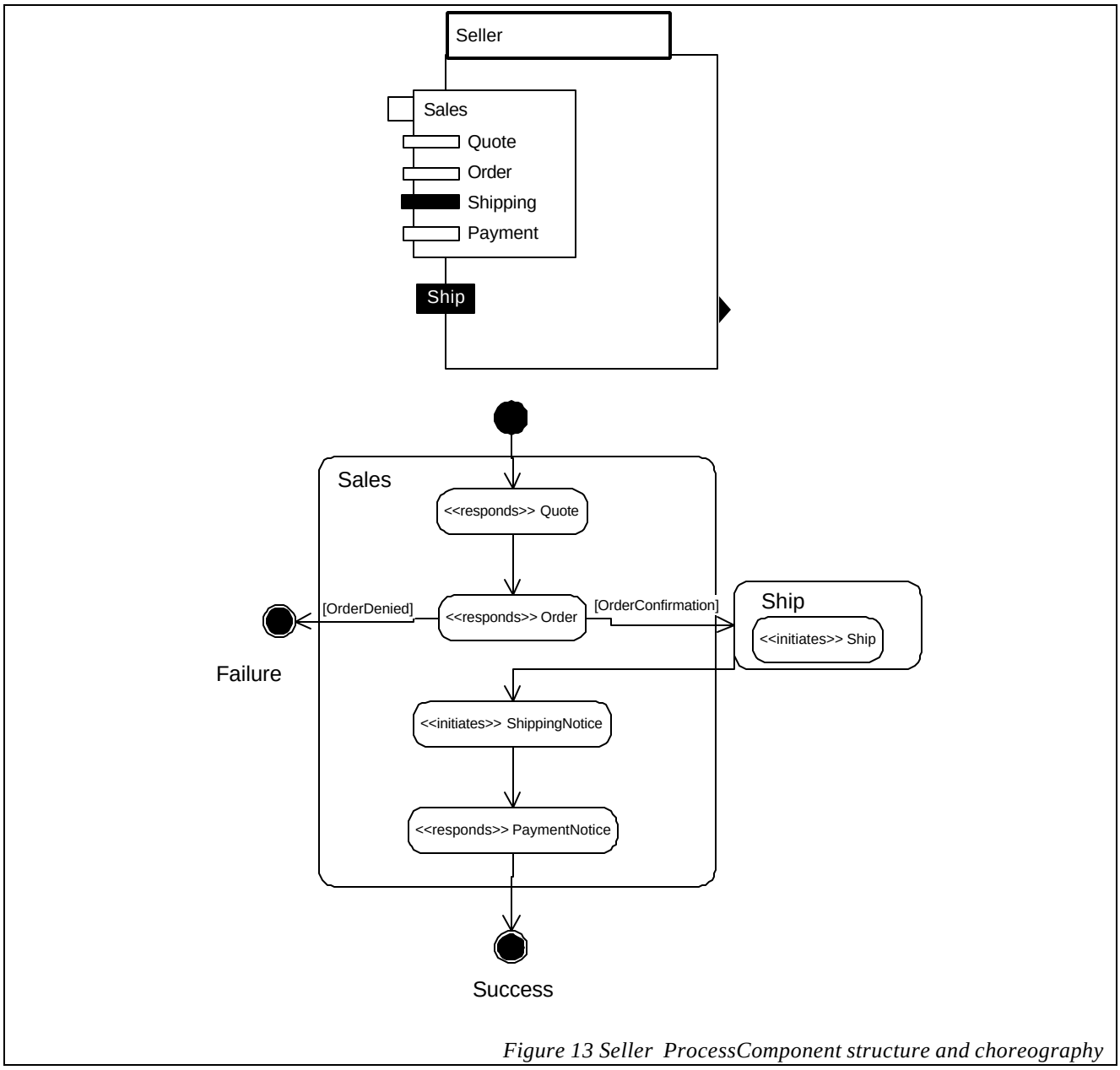

Figure 13 Seller ProcessComponent structure and choreography

Seller ProcessComponent is used in the BuySell CommunityProcess, as ComponentUsage of the same name.

Seller has two ProtocolPort named Sell and Ship.

The Seller responds to interactions through the Sell ProtocolPort according to the Sales Protocol. The Ship ProtocolPort initiates interactions in the Delivery Protocol.

The activity of the Seller ProcessComponent will begin when responding and fully performing the interactions through the Buy Port, according to the used Sales Protocol.

The Failure termination condition of the Sales Protocol is also a Failure termination condition of the choreography of the Seller ProcessComponent.

In the choreography for the Seller ProcessComponent, the interactions through the Ship ProtocolPort, according to the ShipBT Protocol, are inserted as a whole in between two consecutive states of the Sales Protocol in the Sell ProtocolPort.

The choreography of the Seller ProcessComponent is an integration of the choreographies of the Sales and ShipBT Protocols, of the Sell and Ship ProtocolPort. The integration is safely achieved by insertion, as a refinement of a Transition in the Sales Protocol, as two Transitions to and from the inserted Ship PortActivity.

The interactions through the Sell ProtocolPort are integrated with the Ship ProtocolPort, by insertion of the whole ShipBT Protocol, interleaved between two activities of the Sales Protocol. This is a case of safe synthesis, where the constraints and partial ordering of each Protocol are still valid in the synthesized protocol.

The successful termination of the choreography of the Sales Protocol in the Sell ProtocolPort, is also the successful termination of the Seller ProcessComponent.

This structure and choreography fully specify the external contractual obligations and expectations of the Seller ProcessComponent.

No details have been offered, about how the Seller ProcessComponent actually performs its duties, in compliance with the externally observable structure and behavior specified above.

3.4.3 Seller ProcessComponent – internal composition

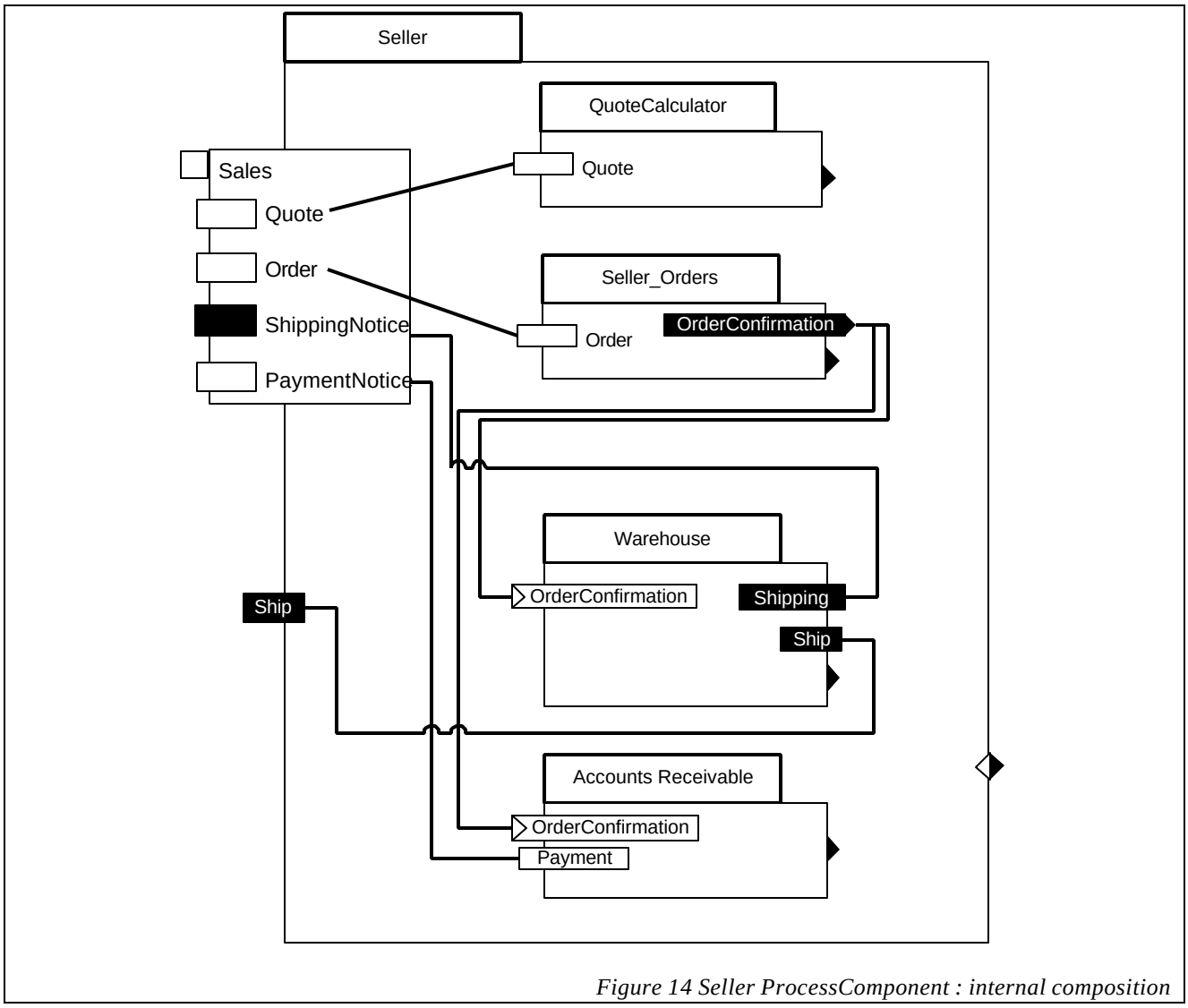

Figure 14 Seller ProcessComponent : internal composition

In the header above, the externally observable structure and choreography have been defined, without revealing any internal details of the Seller ProcessComponent.

When designing a system, that will play the Seller role in a BuySell CommunityProcess, the Seller ProcessComponent will have to be further specified, and its complexity decomposed in smaller units – and recursively – until the resulting ProcessComponent can be directly mapped or implemented to non-CCA artifacts.

The internal de-composition of the Seller ProcessComponent, must comply with the externally observable choreography. If it complies, the Seller may play the role in the BuySell Community Process – and others using the Seller ProcessComponent definition – independently of how the Seller ProcessComponent has been internally defined.

In our example, the Seller ProcessComponent is internally composed by using QuoteCalculator, Seller_Order, Warehouse and AccountsReceivablel components.

The Sell ProtocolPort is rendered expanded, displaying the ProtocolPort of the Sales Protocol, as sub-Port of the Sell ProtocolPort.

The individual sub-ProtocolPort of Sell are delegated or initiated to/from port of sub-component of Seller.

The usage of QuoteCalculator responds to and handles the Quote sub-port of Sell. The QuoteCalculator ProcessComponent has a ProtocolPort using the QuoteBT Protocol, and is therefore compatible for direct delegation from the Quote sub-port of Sell.

Similarly, the Seller_Orders component usage responds to and handles the Order sub-Port of Sell. In addition, the Seller_Orders ProcessComponent has an additional OrderConfirmation outgoing flow, connected to the Warehouse and AccountsReceivable component usages. When Seller_Orders responds an OrderConfirmation, the same OrderConfirmation will be sent to Warehouse and AccountsReceivable.

The Warehouse component usage responds to the OrderConfirmation from the Seller_Orders component, and initiates the interactions of the ShipBT Protocol, forwarded through the Ship ProtocolPort of the container Seller ProcessComponent. After, the Warehouse component initiates the interactions of the ShippingNoticeBT Protocol, through the ShippingNotice sub-Port of Sell.

The AccountsReceivable component usage receives OrderConfirmation from Seller_Orders, and responds to and handles the PaymentNotice sub-port of Sell.

3.4.4 QuoteCalculator ProcessComponent

The QuoteCalculator ProcessComponent has the structure as shown in its component usage in the Seller internal compositions.

QuoteCalculator has a single ProtocolPort responding to the QuoteBT Protocol.

The chorography of QuoteCalculator corresponds to the choreography of the QuoteBT Protocol.

3.4.5 Seller_Orders ProcessComponent

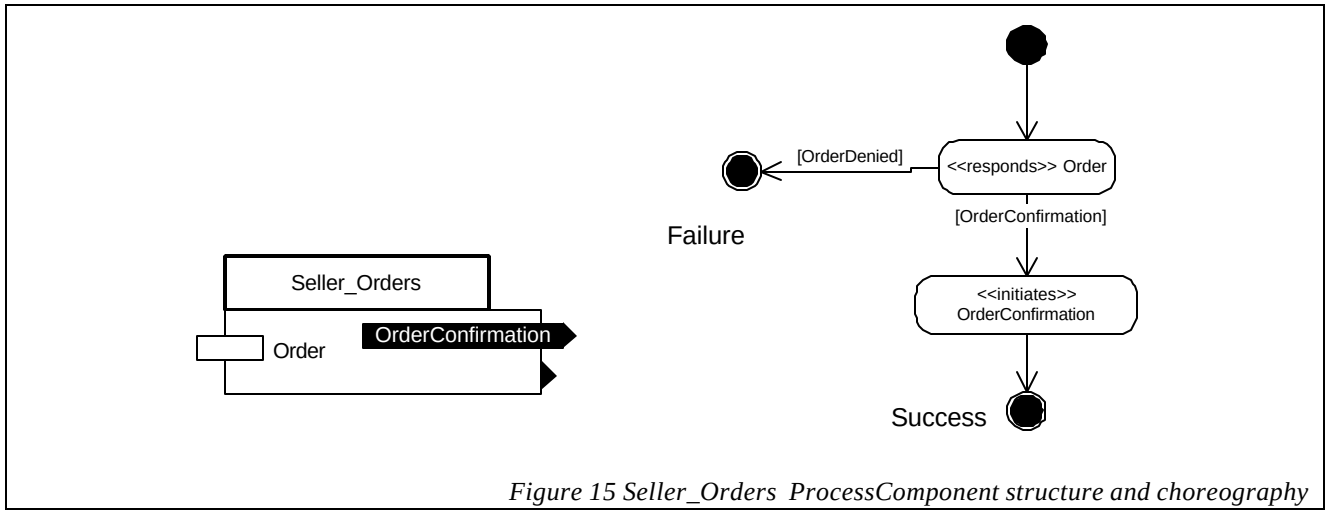

Figure 15 Seller_Orders ProcessComponent structure and choreography

Seller_Orders ProcessComponent responds to interactions of the OrderBT Protocol through the Order ProtocolPort.

The Seller_Orders ProcessComponent has an additional OrderConfirmation outgoing flow. When Seller_Orders responds an OrderConfirmation, the same OrderConfirmation will be sent also through the FlowPort.

3.4.6 Warehouse ProcessComponent

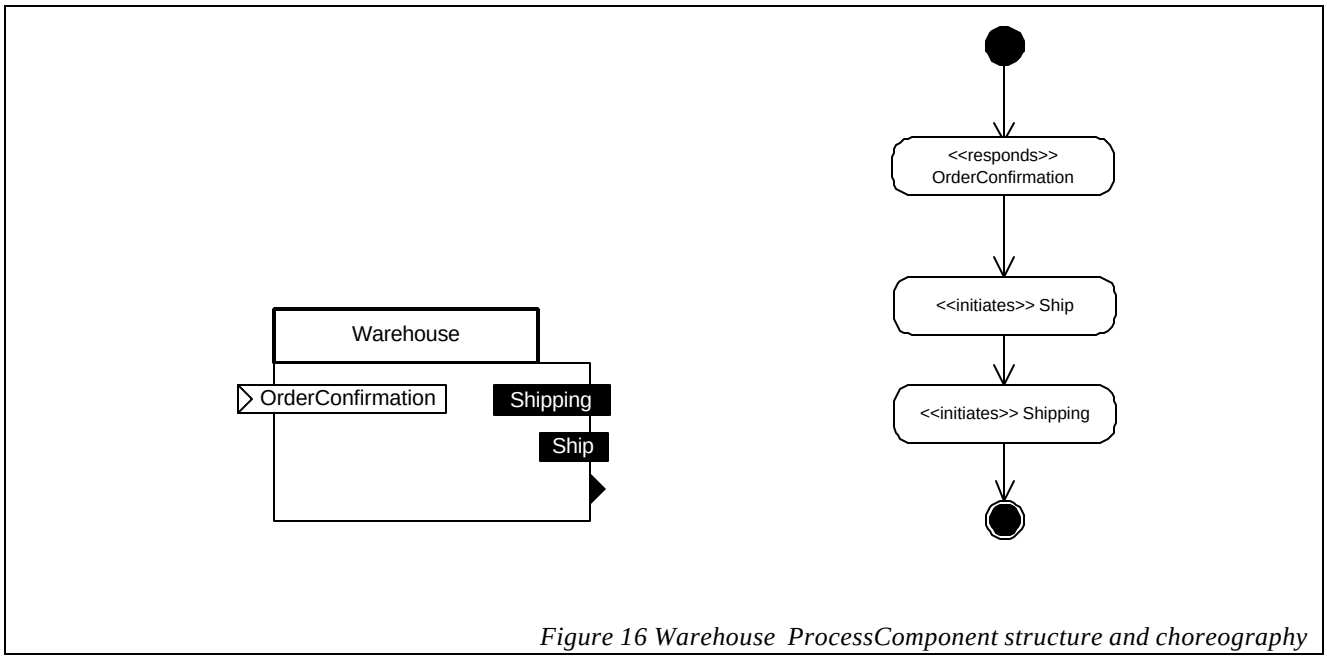

Figure 16 Warehouse ProcessComponent structure and choreography

The Warehouse ProcessComponent receives an OrderConfirmation flow, and initiates the interactions of the ShipBT Protocol, through the Ship ProtocolPort. After, the Warehouse component initiates the interactions of the ShippingNoticeBT Protocol, through the ShippingNotice Port.

3.4.7 AccountsReceivable ProcessComponent

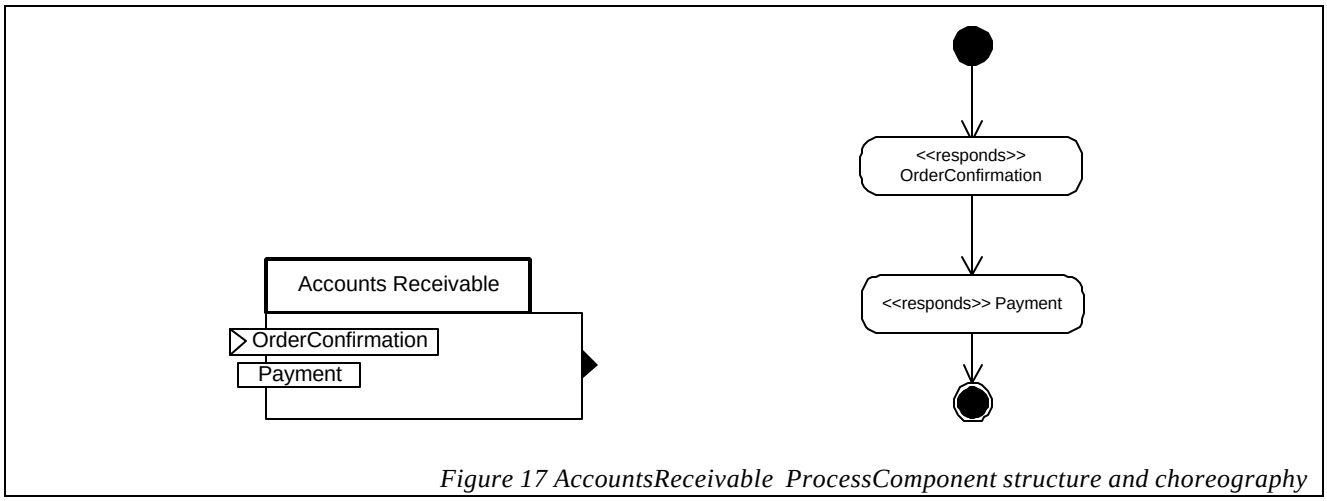

Figure 17 AccountsReceivable ProcessComponent structure and choreography

The AccountsReceivable ProcessComponent receives an OrderConfirmation, and responds to the PaymentNoticeBT Protocol through the Payment ProtocolPort.

3.4.8 Logistics ProcessComponent

Figure 18 Logistics ProcessComponent structure and choreography

Logistics ProcessComponent is used in the BuySell CommunityProcess, as ComponentUsage of the same name.

Logistics has two ProtocolPort named Ship and Delivery.

The Logistics responds to interactions through the Ship ProtocolPort according to the ShipBT Protocol. The Delivery ProtocolPort initiates interactions of the DeliveryBT Protocol.

The activities of the Logistics ProcessComponent will begin by responding and fully performing the interactions through the Ship Port, according to the used ShipBT Protocol.

After this the Logistics ProcessComponent will initiate and fully perform the interactions through the Delivery ProtocolPort.

The Logistics ProcessComponent integrates the ShipBT and DeliveryBT Protocols, by safely synthesizing them in a sequence, where the ShipBT Protocol is fully exercised and completed, before starting the DeliveryBT Protocol.

Glossary

Table 35, below, defines the specialist terms used in this Submission.

Term	Explanation
b2b	Business to Business
b2c	Business to Customer
BFOP	Business Function Object Pattern
CBOP	Common Business Object Patterns Consortium
CCA	Component Collaboration Architecture – a profile for specifying components at multiple levels of granularity
EAI	Enterprise Application Integration
ebXML	XML for Electronic Business
ECA	Enterprise Collaboration Architecture – a set of profiles for making technology independent models of EDOC systems
EDOC	Enterprise Distributed Object Computing – what the submission is all about.
EJB	Enterprise JavaBeans
FCM	Flow Composition Model
RM-ODP	Reference Model of Open Distributed Processing
UML	Unified Modeling Language
VMM	Virtual metamodel: a formal model of a package of extensions to the UML metamodel using UML's own built-in extension mechanisms

Table 35 Glossary of Terms

References

[1] ISO/IEC & ITU-T: Information technology – Open Distributed Processing – Part 1 – Overview – ISO/IEC 10746-1 | ITU-T Recommendation X.901

[2] ISO/IEC & ITU-T: Information technology – Open Distributed Processing – Part 2 – Foundations – ISO/IEC 10746-2 | ITU-T Recommendation X.902

[3] ISO/IEC & ITU-T: Information technology – Open Distributed Processing – Part 3 – Architecture – ISO/IEC 10746-3 | ITU-T Recommendation X.903

[4] ISO/IEC & ITU-T: Information technology – Open Distributed Processing – Enterprise Viewpoint – ITU-T Recommendation X.911 | ISO/IEC 15414

[5] DISGIS Web site: http://www.disgis.com

[6] COMPASS Web site: http://www.compassgl.org

[7] OBOE Web site: http://www.dbis.informatik.uni-frankfurt.de/~oboe/

[8] ISO TC211 Web site: http://www.statkart.no/isotc211/

[9] Open Geodata Consortium Web site: http://www.opengis.org

[10] ISO/IEC JTC1/SC21, Information Technology. Open Systems Interconnection - Management Information Services - Structure of Management Information - Part 7: General Relationship Model, 1995. ISO/IEC 10165-7.

[11] T.Gilb, G.Weinberg. *Humanized Input*. Winthrop Publ., 1977.

[12] H.Kilov, J.Ross. *Information modeling*. Prentice-Hall, 1994.

[13] H.Kilov, L.Cuthbert. A model for document management. *Computer Communications*, Vol. 18, No. 6 (June 1995), pp. 408-417

[14] H.Kilov. *Business specifications*. Prentice-Hall, 1999.

[15] H.Kilov, A.Ash. How to ask questions: Handling complexity in a business specification. In: *Proceedings of the OOPSLA'97 Workshop on object-oriented behavioral semantics (Atlanta, October 6th, 1997),* ed. by H.Kilov, B.Rumpe, I.Simmonds, Munich University of Technology, TUM-I9737, pp. 99-114.

[16] H.Kilov, A.Ash. An information management project: what to do when your business specification is ready. In: *Proceedings of the Second ECOOP Workshop on Precise Behavioral Semantics*, Brussels, July 24, 1998 (ed. by H.Kilov and B.Rumpe). Technical University of Munich, TUM-I9813, pp. 95-104.

[17] H.Kilov, B.Rumpe, I.Simmonds (Eds.). *Behavioral specifications of businesses and systems*. Kluwer Academic Publishers, 1999.

[18] B.Potter, J.Sinclair, D.Till. *An introduction to formal specification and Z*. Prentice-Hall, 1991.

[19] Sun Java Community Process JSR-26 currently under public review, http://jcp.org/jsr/detail/26.jsp

[20] Sun Java Community Process JSR-40 not yet released for public review, http://jcp.org/jsr/detail/40.jsp

[21] MOF 1.3 Specification, OMG document http://cgi.omg.org/cgi-bin/doc?ad/99-09-05

[22] UML Profile for CORBA 1.1 specification, OMG document http://cgi.omg.org/cgi-bin/doc? ptc/01-01-06

[23] Unified Modeling Language Specification, Version 1.4, OMG document http://cgi.omg.org/cgi-bin/doc?ad/01-02-13

[24] XMI 1.1 Specification, OMG document http://cgi.omg.org/cgi-bin/doc?ad/99-10-02

[25] Unified Modeling Language Specification, Version 1.3, June, 1999 http://cgi.omg.org/cgi-bin/doc?ad/99-06-08

[26] Desmond F. D'Souza, Alan Cameron Wills. Objects, Components, and frameworks with UML: The Catalysis Approach. Reading, Mass., Addison-Wesley, 1999.

[27] Martin Fowler. M. Analysis Patterns: Reusable Object Models. Reading, Mass., Addison-Wesley, 1997.

[28] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[29] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Development Process.* Addison-Wesley, Reading, Mass., 1999.

[30] OMG, Model Driven Architecture – under development

[31] Trygve Reenskaugh, Per Wold and Odd Arild Lehne. *Working with Objects : the OORAM Software Engineering Method* 1996 Manning Publications Co. 1996

[32] Bran Selic, Garth Gullekson and Paul T. Ward *Real-Time Object-Oriented Modeling*. John Willey & Sons, Inc. 1994