A UML Profile for Enterprise Distributed Object Computing

Joint Revised Submission

Part IIIa – Component Collaboration Architecture Profile

Version 0.92

26 February 2001

Submitted by:

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

Supported by:

Hitachi
SINTEF
Netaccounts

Changes on OMG Document Number: ad/2001-02-19

Foreword

The ECA UML for EDOC Profile Submission

The ECA UML for EDOC Profile Submission is a specification for a UML Profile for Enterprise Distributed Object Computing, 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).

Co-submitting Companies

This submission is prepared by the following companies:

?? CBOP
?? Data Access Technologies
?? EDS
?? Fujitsu
?? Iona Technologies
?? Open-IT
?? Sun Microsystems

Supporting companies are:

?? Hitachi
?? Netaccounts
?? SINTEF

Status of this document

This document is the second iteration in a submission process that commenced in October 1999, when initial submissions were made. At that time it was hoped that a single joint submission team could be formed to prepare a single Final submission by this time. Regrettably, because the requirements of the RFP are very wide and complex, it has not been possible to achieve that aim, and although considerable effort has been expended to consolidate all the ideas and requirements of the submitting team, it is acknowledged that there is still some work required to reduce conceptual overlap and produce a complete and internally consistent submission.

It is the faith of the submission team that this can be done, in collaboration with other UML for EDOC submitters not members of this submission team, in the time between review of this revised submission and the deadline for a Final submission that is agreed at the ADTF meeting in Irvine in February 2001.

The set of documents is acknowledged to be incomplete at the current issue. In particular:

A major element of the submission, the Distributed Component Profile (Part IV) is not included in this set, but is published as a separate submission, submitted by a set of

companies largely the same as the ECA consortium. This document may be found at ad/2001-02-20. The DCP details how to utilize the UML to specify a particular kind of component called a Distributed Component or DC. A DC can usefully be characterized as being:

?? a pluggable autonomous software artifact that has a “distributed” interface

?? represents a single concept

?? is intended to be deployed as a managed run-time artifact

?? when implemented and deployed, will typically execute in a single address space.

It is the intention of the submitters to prepare a fully worked example that uses as much of the profile as possible. This will form Part VI of the submission, but the work has not yet been completed.

Guide to the Submission

This submission is divided into the following parts as illustrated by the figure below:

Part I is the formal response to the submission as required by the RFP. Part 1 calls up the remaining parts in the set to create a complete submission.

Part II describes the Enterprise Collaboration Architecture (ECA) which is the framework for system specification using the EDOC Profile. It provides a detailed rationale for the modelling choices made and describes how the other elements in the submission, detailed in Part III, 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 behavior is modified and tuned to meet the changing business environment in which it will work.

Part III contains the detailed profile specifications for the modelling 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 Entity profile, which describes a set of UML extensions that may be used to model entity objects;

?? the Event profile, which describes a set of UML extensions that may be used on their own, or in combination with the other EDOC elements described in Part III, to model event driven systems;

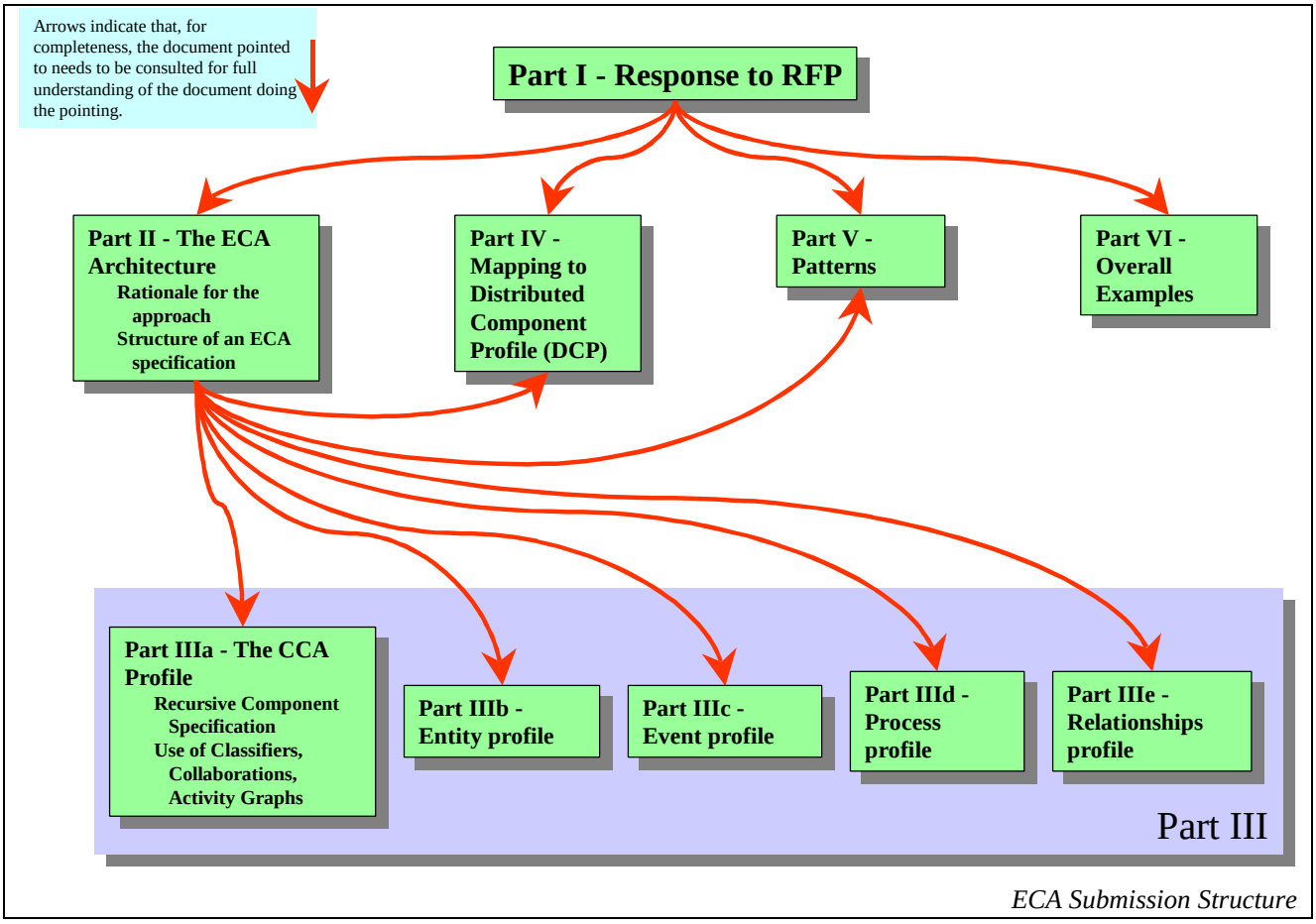
?? the Process profile, which describes a set of UML extensions that may be used on their own, or in combination with the other EDOC elements described in Part III, to model system behavior in the context of the business it supports;

?? 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.

Part IV is a mapping of the ECA concepts to the Distributed Component Profile (DCP).

Part V 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.

Part VI details worked examples illustrating all aspects of the Profile. (Note that this Part is not complete and not included in this Revised Submission.)



ECA Submission Structure

Table of Contents

Foreword ii
The ECA UML for EDOC Profile Submission ii
Co-submitting Companies ii
Status of this document ii
Guide to the Submission iii

Table of Contents v

Table of Figures vii

1. Introduction 1
1.1 Document Status 1
1.2 Logical Meta-Model & UML Profile 2
1.3 CCA Notation 2

2. Rationale 3
2.1 Problems to be solved 3
2.2 Approach 6
2.3 Conceptual Framework 9

3. CCA Logical Meta-Model Specification 11
3.1 CCA concept – UML Stereotype – UML base 11
3.2 UML Stereotype – Tagged Values 12
3.3 Enumeration values 13
3.4 Process Component Definition 13
3.5 Protocol Specification 21
3.6 Component Realization 27
3.7 Composition 32
3.8 Choreography 43
3.9 Document Model 53
3.10 Model Management 59
3.11 Combined Model Diagram 63

4. Notation 64
4.1 Process Component Specification Notation 64
4.2 Protocol Notation 65
4.3 Composite Component Notation 66
4.4 Primitive Component Notation 68
4.5 Community Process Notation 68
4.6 Composition Notation 68
4.7 Choreography Notation 69
4.8 Data Model Notation 69
4.9 Model Management Notation 69
4.10 Data Manager Notation 69

5. UML Profile Specification 71
5.1 Introduction 71
5.2 Relationship with Conceptual Meta-Model 71
5.3 Choice of UML elements 71
5.4 Profile structure 72
5.5 ComponentSpecification «profile» Package 73
5.6 Protocol «profile» Package 79
5.7 ComponentRealization «profile» Package 85
5.8 Composition «profile» Package 87
5.9 Choreography «profile» Package 94
5.10 DocumentModel «profile» Package 101
5.11 High-level ActivityGraph of a Composition 102
5.12 Common «profile» Package 103
5.13 Owners «profile» Package 106

6. Constraints (OCL) 110
6.1 Invariant Constraints (OCL) 110
6.2 Definition Constraints (OCL) 111

7. Samples 116
7.1 CCA Notation 116
7.2 UML Notation 128
7.3 UML-RT Notation 141

8. Proof of correctness 145

9. References 155

Table of Figures

Figure 1: Structure and dependencies of the CCA Conceptual Meta-Model Packages 9
Figure 2: ComponentDefinition Conceptual Meta-Model 14
Figure 3: Protocol Conceptual Meta-Model 21
Figure 4: ComponentRealization Conceptual Meta-Model 28
Figure 5: Composition Conceptual Meta-Model 33
Figure 6: Choreography Conceptual Meta-Model 44
Figure 7: DocumentModel Conceptual Meta-Model 54
Figure 8: ModelManagement Conceptual Meta-Model 60
Figure 9: Combined Conceptual Meta-Model 63
Figure 10: ProcessComponent specification notation 64
Figure 11 Protocol Notation (1) 65
Figure 12: Protocol notation (2) 66
Figure 13: Composite Component notation 67
Figure 14: PrimitiveComponent notation 68
Figure 15: CommunityProcess notation 68
Figure 16: DataModel notation 69
Figure 17: DataManager notation 69
Figure 18: Structure and dependencies of the CCA «profile» Packages 73
Figure 19: Class Diagram of the Virtual metamodel for ComponentSpecification «profile» Package 74
Figure 20: Class Diagram of the Virtual metamodel for Protocol «profile» Package 79
Figure 21: Class Diagram of the Virtual metamodel for ComponentRealization «profile» Package 85
Figure 22: Class Diagram of the Virtual metamodel for Composition «profile» Package 88
Figure 23: Class Diagram of the Virtual metamodel for Choreography «profile» Package 94
Figure 24: Class Diagram of the Virtual metamodel for DocumentModel «profile» Package 101
Figure 25: Class Diagram of the Virtual metamodel for Common «profile» Package 103
Figure 26: Class Diagram of the Virtual metamodel for Owners «profile» Package 107
Figure 27 Sample CompositeData definition (CCA) 116
Figure 28: Sample Choreographed Protocol (CCA) 117
Figure 29: Sample Choreographed RequestReplyProtocol (CCA) 118
Figure 30: Sample Choreographed FlowProtocol (CCA) 118
Figure 31: Sample Choreographed FlowProtocol (CCA) 119
Figure 32: Sample Choreographed Protocol with subProtocols (CCA) 119
Figure 33: Sample ProcessComponents (CCA) 120
Figure 34: Sample ProcessComponents (CCA) - will be used in the ComposedComponent example 121
Figure 35: Sample Composition as a CommunityProcess. (CCA) 121
Figure 36: ContextualBinding (in CommunityProcess) (CCA) 122
Figure 37: ComposedComponent (CCA) 124
Figure 38: ProcessComponent for example on Choreography of ProcessComponent (CCA) 126
Figure 39: Choreography of ProcessComponent – with sub-Protocols (CCA) 126
Figure 40: Choreography of ProcessComponent (CCA) 127
Figure 41: High Level ActivityGraph of Composition (CCA) 128
Figure 42: Sample CompositeData definition (UML) 128
Figure 43: Sample Protocol (UML) 129
Figure 44: SampleRequestReplyProtocol (UML) 129
Figure 45: Sample FlowProtocol (UML) 130
Figure 46: Sample Choreographed Protocol (UML) 130
Figure 47: Sample Choreographed RequestReplyProtocol (UML) 131

Figure 48: Sample Choreographed FlowProtocol (UML) 131
Figure 49: Sample Protocol, RequestReplyProtocol, FlowProtocol (UML Collaboration view) 132
Figure 50: Sample Protocol with SubProtocols (UML) 133
Figure 51: Sample Choreographed Protocol with exploded SubProtocols (CCA) 134
Figure 52: Sample Protocol with SubProtocols (UML Collaboration view) 135
Figure 53: Sample ProcessComponents, with PropertyDefinitions, and ProtocolPorts (UML) 135
Figure 54: Some components for the ComposedComponent example (UML) 136
Figure 55: Sample Composition as a CommunityProcess (UML) 137
Figure 56: Sample Composition as a CommunityProcess, (UML Collaboration view) 137
Figure 57: ContextualBinding on CommunityProcess (UML) 138
Figure 58: ContextualBinding on CommunityProcess, compact form (UML) 139
Figure 59: ComposedComponent (UML) 140
Figure 60: Composition of ComposedComponent (UML Collaboration view) 141
Figure 61: Sample Protocol, RequestReplyProtocol, FlowProtocol (RT) 142
Figure 62: Sample Protocol with messages manual copied from SubProtocols (RT) 142
Figure 63: Sample ProcessComponents , Class view (RT) 142
Figure 64: Buyer and Seller ProcessComponents , Buyer, Structure Diagrams (RT) 142
Figure 65: Some components for the ComposedComponent example, Class view (RT) 143
Figure 66: Order_seller, Quote_seller_Payment_seller, Shipping_seller: Structure Diagrams (RT) 143
Figure 67: Sample Composition as a CommunityProcess. Structure Diagram (RT) 143
Figure 68: Specialized Composition (RT) 143
Figure 69: ComposedComponent (RT) 144
Figure 70: CompositeData (M1s) 145
Figure 71: Sample Protocol, RequestReplyProtocol, FlowProtocol (M1s) 146
Figure 72: Sample Protocol with SubProtocols (M1s) 147
Figure 73: Sample ProcessComponents, with PropertyDefinitions, and ProtocolPorts (M1s) 148
Figure 74: Some components for the ComposedComponent example (M1s) 149
Figure 75: Sample Composition as a CommunityProcess (M1s) 150
Figure 76: Specializing Composition with ContextualBinding (as a CommunityProcess) (M1s) (as Collaboration) 151
Figure 77: ComposedComponent (M1s) 152
Figure 78: Choreography of a Protocol (M1s) 153
Figure 79: Choreography of a Protocol with sub-Protocols (M1s) 154
Figure 80: High Level Activity Graph of a Composition (M1s) 154

1. Introduction

This document specifies the Component Collaboration Architecture (CCA). The CCA is a key part of the Enterprise Collaboration Architecture (ECA), a response to OMG RFP for a UML profile for Enterprise Distributed Object Computing (EDOC), and is referenced by the response to the OMG RFP for a UML profile for Enterprise Application Integration (EAI).

The CCA specification details how to utilize the UML to specify, at multiple levels of granularity, components that collaborate to fulfill some purpose. As a specification it is intended for analysts, designers, modelers and tool builders already familiar with the UML.

While initially targeted as a core part of the UML profiles for EDOC and EAI, the CCA is a general-purpose architecture for recursive composition and decomposition of component-based information systems, which may be applied to many domains.

The CCA is based, in part, on research funded by the National Institute of Standards, Advanced Technology Program in a co-operative agreement with Data Access Technologies.

Authors

The Primary authors of this document are:

?? Cory Casanave – Data Access Technologies

?? Antonio Carrasco-Valero – Data Access Technologies

In addition valuable input was received from all members of the ECS submitters team.

1.1 Document Status

This is a draft document. Several issues still exist with the profile and with how it uses UML. The document is not complete (I.E. UML OCL has not been done) and there may be inconsistencies to resolve and it certainly needs editing. This draft is intended for the RFP submission teams working with it.

The purpose of this draft is to validate CCA against the requirements of EDOC and EAI, provide a basis for moving ahead with these more domain specific profiles and to solicit input and participation in its refinement.

At some point we expect to do an overall “naming” review to get the CCA terms in-line with EDOC/EAI and general intuitiveness.

1.1.1 Relation to EDOC & ECA

The CCA is a part of, but not the entire, profile for EDOC and the ECA. There are separate specifications for the Information model, Process model, Events and Patterns.

These sub-profiles are brought together in the ECA part of EDOC and it is expected that the ECA will reference and refine CCA for specific viewpoints and different levels of granularity.

1.2 Logical Meta-Model & UML Profile

The specification of this profile contains a logical Metamodel. This metamodel shows the logical structure of the concepts used in CCA in a MOF compliant structure suitable for custom tools. The UML profile as a set of stereotypes, tagged values and constraints are shown in relation to this logical model, providing the capability for off-the-shelf tools to support CCA.

Most elements of the CCA Meta-Model directly correspond to UML elements or are logical subtypes of them. These elements are defined independently in the CCA model and then their relationship to UML elements is shown. When CCA and UML Meta-Model elements have the same name it may be assumed that have the same semantics.

1.3 CCA Notation

CCA models may utilize standard UML notation or a CCA specific notation. Current off-the-shelf UML tools may use the standard UML notation while CCA aware tools may use the CCA notation, which is somewhat more compact and intuitive.

2. Rationale

2.1 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 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 rational 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.

2.1.2 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.

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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, Bizalk. 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.

2.1.7 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.

2.2 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.

The UML is a standard that has become accepted as the way to model systems at many levels and for a variety of purposes. As such it is ideal for the CCA. The UML is designed to be specialized for specific purposes using a mechanism called a “profile”. A profile uses the extension mechanisms of UML to focus on a specific modeling requirement or paradigm. In the case of the CCA this is recursive decomposition and assembly of parts of an information system.

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 Process Component. In some cases Process Components 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, distributed objects, real-time etc.

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

The CCA describes how Process Components 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, Process Components exposed on the Internet will require features of security and distribution, while more local Process Components will only require a way to communicate.

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

2.2.1 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 components are processing components, ones that collaborate with other CCA components within a CCA composition. CCA components can be used to build other CCA components 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 components and composition are orthogonal to these concepts. A technology component, such as an EJB may be the implementation platform for a CCA component.

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.

All references to component in this document are specific to the CCA component and composition model.

2.2.2 Process Component Libraries

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

2.2.3 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 Process Components 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 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 mappings from the CCA Profile to various engineering and technology options, see Part IV of this submission.

2.2.4 Specification Vs. Methodology

The CCA provides a way to specify a system in terms of a hierarchical structure of Communities of Process Components 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.

2.2.5 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 3achieve greater simplicity and economy of expression.

2.3 Conceptual Framework

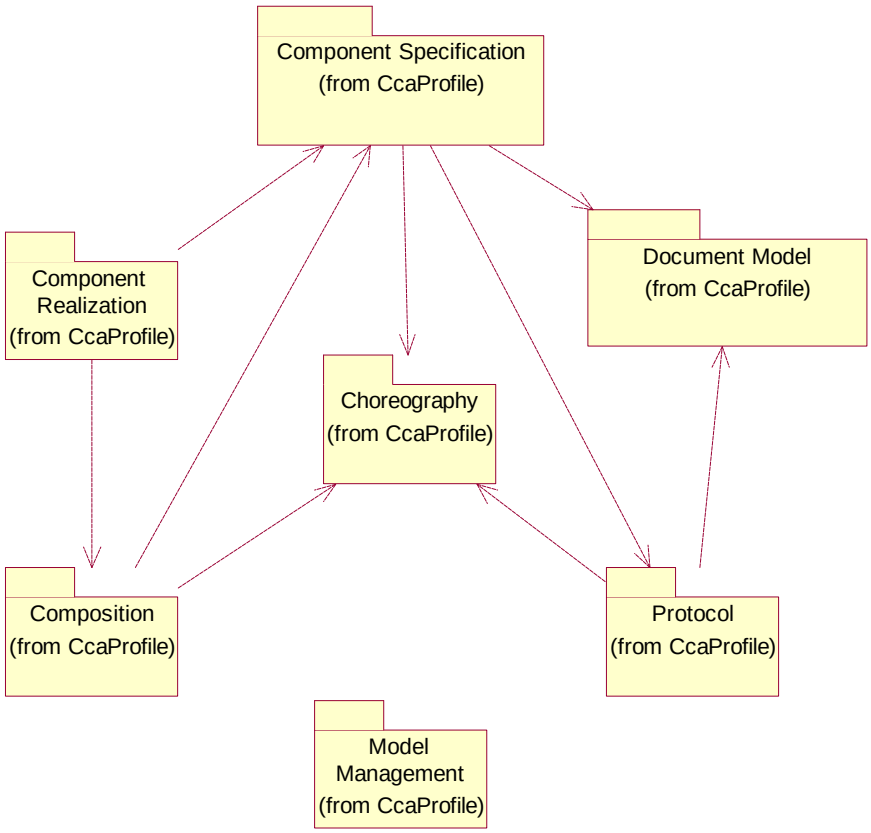


Figure 1: Structure and dependencies of the CCA Conceptual Meta-Model Packages

2.3.1 Process Component 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".

2.3.2 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.

2.3.3 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 **abstract component** 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**.

2.3.4 Composition

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

Compositions can also include a **choreography** of how the components used work together, which should execute when.

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.

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

2.3.5 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 process components.

2.3.6 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.

3. CCA Logical Meta-Model Specification

3.1 CCA concept – UML Stereotype – UML base

This table summarizes the correspondence between elements of the CCA Conceptual Meta-Model, the Stereotypes in the UML Profile, and the baseClasses of the Stereotypes.

Package	**Conceptual Meta-Model**	**Stereotype**	**UML base class**	**Comment**
ComponentSpecification	ProcessComponent	ProcessComponent	Subsystem	
	Port	Port	Class	abstract
	ProtocolPort	ProtocolPort	Class	
	FlowPort	FlowPort	Class	
	PropertyDefinition	Property	Attribute	
		Granularity		Enumeration
Protocol	Protocol	Protocol	Subsystem	
	RequestReplyProtocol	RequestReplyProtocol	Subsystem	
		FlowProtocol	Subsystem	for FlowPort
	ProtocolRole	ProtocolRole	Class	
		FlowRole	Class	For FlowPort
	Interaction	-	-	abstract
	ProtocolMessage	ProtocolMessage	Reception	
	SubProtocol.one role	SubProtocolRole	Class	+Generalization
ComponentRealization	PrimitiveComponent	PrimitiveComponent	Subsystem	
	ComposedComponent	ComposedComponent	Subsystem	
	CommunityProcess	CommunityProcess	Subsystem	
Composition	Composition	Composition	Subsystem	
	ComponentUsage	ComponentUsage	Subsystem	
	PortUsage	PortUsage	Class	
	PortProxy	PortProxy	Class	
	ConnectionRole			abstract
	Connection	Connection	Association	
	PropertyValue	Property	Attribute	(same as PropertyDefinition)
	ContextualBinding	ContextualBinding	Binding	
Choreography	Choreography	Choreography	ActivityGraph	

	State			abstract
	Transition	ChoreographyTransition	Transition	
	Start	Start	Pseudostate-initial	
	TerminateSuccess	TerminateSuccess	FinalState	
	TerminateFailure	TerminateFailure	FinalState	
	Split	Split	Pseudostate-fork	
	Join	Join	Pseudostate-join	
	MessageStep	MessageStep	Transition	with SendAction 'effect' or SignalEvent 'trigger'
	SubProtocolStep	SubProtocolStep	ActionState	
	SubStep	SubStep	SubActivityState	
DocumentModel	CompositeData	CompositeData	Class	
Common		ProtoPort	Class	
		ProtoComponent	Subsystem	
		PropertyHolder	Class	
		Property	Attribute	
Owners		PortOwner	Subsystem	
		ComponentOwner	Subsystem	
		ConnectionOwner	Subsystem	
		ProxyOwner	Subsystem	
		PropertyHolderOwner	Subsystem	
		CompositionOwner	Subsystem	
		MessageOwner	Class	
		PortNester	Class	

3.2 UML Stereotype – Tagged Values

This table summarizes the taggedValues defined for the Stereotypes.

Package	**Stereotype**	**taggedValue**	**type**	**Comment**
Protocol	ProtocolRole	initiator	Boolean	
	ProtocolMessage	postCondition	Choreography::Status	
ComponentSpecification	ProcessComponent	granularity	Granularity	
		persistent	Boolean	
	ProtocolPort	synchronous	Boolean	
		transactional	Boolean	
		multiple	Boolean	

Composition	Connection	protocolScope	Protocol::Protocol	
		messageScope	Protocol:: ProtocolMessage	
ComponentRealization	PrimitiveComponent	implementationType	String	
		implementationLocation	String	
Choreography	SubStep	scope	Choreographed	
	Transition	precondition	Choreography::Status	

3.3 Enumeration values

This table summarizes the values of the defined enumeration types.

Package	**Enumeration**	**values**	**Comment**
ComponentSpecification	Granularity	Program Owned Shared	
Choreography	Status	Success TimeoutFailure TechnicalFailure BusinessFailure AnyFailure Any	
	DirectionKind	Sends Receives	

3.4 Process Component Definition

The ProcessComponent definition specifies the externals of a ProcessComponent, i.e. its contract with other ProcessComponents. ProcessComponent specification relies on the specification of protocols, choreographs and documents, which are documented in other sections. A diagram relating all of the major model elements may be found on page **Error! Bookmark not defined.**.

3.4.1 Conceptual Meta-Model

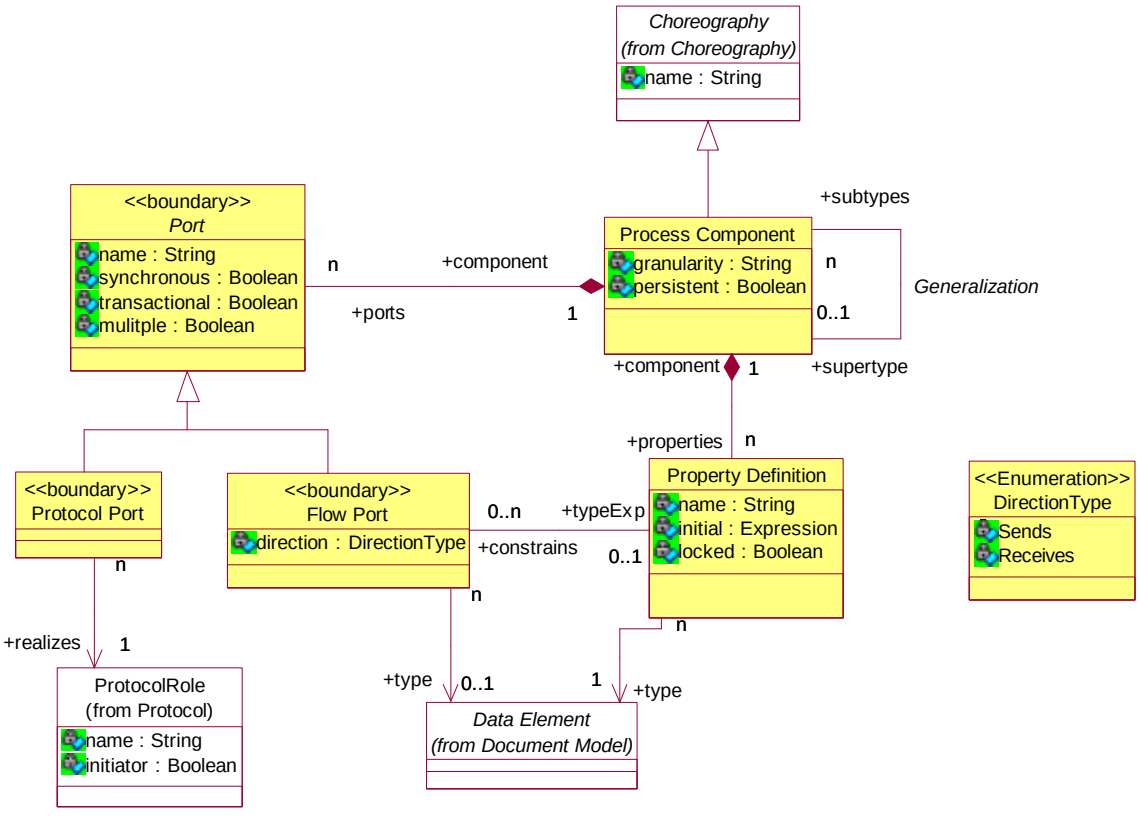


Figure 2: ComponentDefinition Conceptual Meta-Model

3.4.1.1 Summary

A **Process Component** represents the contract for a component which performs actions. A Process Component may realize a set of **Port**s for interaction with other Process Components. The Process Component defines the external contract of the component in terms of ports and a **Choreography** of port actions (sending or receiving messages or initiating sub-protocols). Process components specify the externals of the component that may be realized as concrete **primitive components** or **composite components** (See Component Realization).

The contract of the process component is realized via **ports**. A port defines a point of interaction between process components. The simpler form of port is the **Flow Port**, which may produce or consume a single **data type**. More complex interactions between components use a **Protocol Port**, which refers to a **protocol role** – one end of a more complex interaction between two components (see "Protocol Specification")

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

The specification of the process component may include **Choreography** to sequence the actions of multiple ports and their associated actions. The actions of each port may be **Choreographed**. Choreography is defined in its own section.

A process component may have a **supertype**. One common use of supertype is to place abstract process components 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..

3.4.2 Model Elements

3.4.2.1 Process Component

Extends

Choreography (Indicating that a Choreography of port actions may be specified)

Owned By

Package

Semantics

A Process Component represents an active processing component – it does something. A Process Component may realize a set of Ports for interaction with other Process Components and it may be configured with properties. An instance of process component represents an abstract component, one with no defined implementation. The subtypes of Process Component: Primitive Component & Composed Component provide implementation detail for concrete components. Direct instances of Process Component are abstract.

Each component realizes a set of ports for interaction with other components and has a set of properties that are used to configure the component when it is used.

The order in which actions of the components ports do something may be specified using Choreography.

Elements

Ports (any number)

"Ports" is the set of Ports on the Process Component. Each port provides a connection point for interaction with other components and realizes a specific protocol. The protocol may be simple and use a "flow port" or the protocol may be complex and use a "Protocol Port". If allowed by its protocol, a port may send and receive information.

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

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

A subtype of a component may be substituted for its supertype.

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.

Granularity

<<More here from Oliver Sims>> A string which defines the scope in which the component operates. The base 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.

Persistent

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

A CCA ProcessComponent is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem, and a Stereotype of Foundation::Core::Class named "PropertyHolder", and the «enumeration» "Granularity". See details in section 5 "UML Profile Specification", subsection "ComponentSpecification «profile» Package", headings "ProcessComponent" and "Granularity" and subsection "Common «profile» Package", heading "PropertyHolder".

3.4.2.2 Port

Extends

Choreographed (Indicating that a port may be Choreographed by the process component's Choreography)

Owned By

Process Component

Semantics

A port realizes a simple or complex protocol for a process component. Port is abstract and has two subtypes; Protocol Port and Flow Port. A Flow Port realizes a simple data flow into or out of a component and protocol port realizes a more complex protocol. All interactions with a process component 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

Elements

Component (Exactly One)

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

Transactional

Indicates that interactions with the component are transactional & atomic (in most implementations this will required 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.

Synchronous

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

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 realizes.

Multiple

Allows multiple collaborators of a compatible protocol to be attached to the port.

UML

A CCA Port is represented in the UML profile for CCA, as an abstract Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "ComponentSpecification «profile» Package", heading "Port".

3.4.2.3 Protocol Port

Extends

Port

Owned By

Process Component

Semantics

A protocol port is a process component port which realizes a protocol role, which is defined as part of a protocol (See protocol package). A protocol port is used for potentially complex two-way interactions between components, such as is common in B2B protocols. By realizing one of the two protocol roles of a protocol, the protocol port takes on the responsibility of sending and receiving messages as defined in that protocol.

Elements

Realizes

The protocol role realized by this port on behalf of the component.

UML

A CCA ProtocolPort is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "ComponentSpecification «profile» Package", heading "ProtocolPort".

3.4.2.4 Flow Port

Extends

Port

Owned By

Process Component

Semantics

A Flow Port is a process component port which realizes a data flow in our out of the port on behalf of the component.

Elements

type

The type of information sent or received by this port. If not set the port may send or receive any type of information, which is useful for generic components .

typeExp

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 *typeExp* are both set then the property expression must return the name of a subtype of *type*.

direction

The port may send or receive information of the appropriate type. If information is sent out, direction has a value of "sends". If information is received, direction has a value of "receives".

UML

A CCA FlowPort is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "ComponentSpecification «profile» Package", heading "FlowPort".

3.4.2.5 Property Definition

Extends

None

Owned By

Process Component

Semantics

Since components are designed for reuse in a variety of circumstances they may require configuration when used. Property definitions provide a way do specify the configurable properties of a component including the name, type and default value of each. When the component is used in a composition the property can be set, specializing it for each use. Specific implementation technologies may also allow runtime or deployment time configuration of properties.

Elements

component

Component for which this is s a property.

type

Type of information in the property.

constrains

Flow ports for which the property configures their type. If the cardinality of "constrains" is greater than zero, the property must return a type name.

name

Name of the property.

initial

Expression returning the default value of the property.

locked

If locked is true, the value may not be change in uses of the component.

Constraints

If the cardinality of "constrains" is greater than zero, the property must return a type name.

UML

A CCA PropertyDefinition is modeled in UML as a Stereotype, named "PropertyDEfinition", of Foundation::Core::Attribute. See details in section 5 "UML Profile Specification", subsection "ComponentSpecification «profile» Package", heading "PropertyDefinition".

3.5 Protocol Specification

A protocol is a choreography of interactions between two protocol roles. Components realize a specific protocol role using a protocol port.

3.5.1 Conceptual Meta-Model

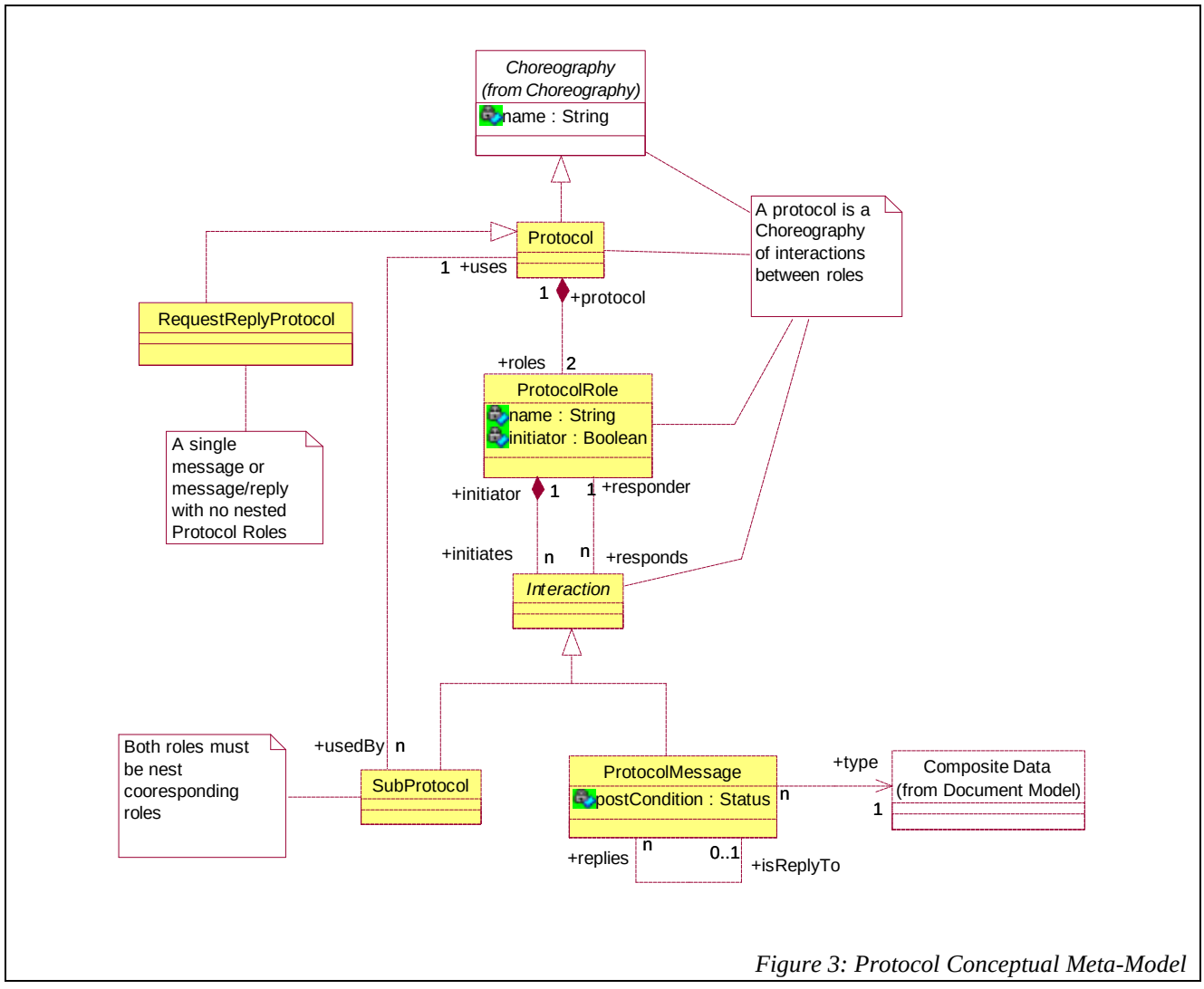


Figure 3: Protocol Conceptual Meta-Model

3.5.1.1 Summary

A **protocol** defines a conversation between two parties, each of which is represented by a **Protocol Role**. One protocol role is the initiator of the conversation and the other the responder. However, after the conversation has been initiated, individual interactions may by initiated by either party.

Within the protocol, one of the protocol roles sends a **Protocol Message** which may or may not have direct **replies**. While multiple kinds of replies are allowed, only one may be used as the reply for any particular message instance.

A protocol may also utilize **Sub Protocols**. This allows one protocol to use another (or multiple others). For example, a sale protocol may use order, invoice and payment protocols.

A **Request Reply Protocol** is a constrained form of protocol patterned after the ebXml "Business Transaction". Its intent is to model a single message and reply as a reusable element. A Choreography is not required since it is pre-defined by the initiation and reply pattern – similar to an asynchronous method call.

3.5.2 Model Elements

3.5.2.1 Protocol

Extends

Choreography (Indicating that a Choreography of interactions (messages and sub-protocols) ay be specified)

Owned By

Package

Semantics

A protocol specifies two protocol roles which interact using messages and sub-protocols. The protocol specifies all the potential interactions and the choreography of those interactions.

Elements

roles

The two protocol roles participating in the protocol.

usedBy

The set of SubProtocols which use this protocol role as a sub-protocol.

Constraints

The initiating role must initiate the first message.

UML

A CCA Protocol is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem. See details in section 5 "UML Profile Specification", subsection "Protocol «profile» Package", heading "Protocol".

3.5.2.2 Protocol Role

Extends

None

Owned By

Protocol

Semantics

A protocol role represents one “end” of a two-way conversation. Each role (the initiator and the responder) may send and receive messages as part of the conversation.

A protocol role is realized by a protocol port, which enables a component to participate in the conversation with another component. The same protocol role may be realized by multiple protocol ports, even on the same component.

Elements

protocol

The protocol for which this is a role.

initiates

The set of interactions (messages and sub-protocols) initiated by this role.

responds

The set of interactions (messages and sub-protocols) received by this role.

name

The name of the role.

initiator

The role which initiates the first interaction, the “client”.

UML

A CCA ProtocolRole is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "Protocol «profile» Package", heading "ProtocolRole".

3.5.2.3 Interaction

Extends

Choreographed (indicating that interactions can be choreographed by the protocol).

Owned By

Protocol Role

Semantics

Interaction is an abstract class representing a portion of a conversation between two protocol roles which are the “initiator” and “responder”. The interaction may be Choreographed by the Protocols Choreography.

Elements

Initiator

The role initiating the conversation fragment, I.E. seining the initial message.

Responder

The role responding to the conversations fragment, I.E. receiving the message.

Constraints

The initiator and responder are both owned by the same protocol.

UML

A CCA Interaction is abstract. Only the concrete specializations of Interaction correspond to UML Stereotypes.

3.5.2.4 ProtocolMessage

Extends

Interaction

Owned By

ProtocolRole

Semantics

The specification that a message of a given type can be sent between the initiator and the responder roles.

Elements

Type

The type of information carried by the protocol message.

Replies

The list of messages which are potential replies to this message

IsReplyTo

The message, if any, that this is a reply to.

PostCondition

The success or failure condition implied by the message.

Constraints

A reply cannot have replies.

UML

A CCA ProtocolMessage is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::Common Behavior::Reception, referencing a Behavioral Elements::Common Behavior::Signal, with an Foundation::Core::Attribute of type Class stereotyped as CompositeData, or a DataType or a User defined DataType or an Enumeration. See details in section 5 "UML Profile Specification", subsection "Protocol «profile» Package", heading "ProtocolMessage".

3.5.2.5 SubProtocol

Extends

Interaction

Owned By

ProtocolRole

Semantics

A protocol may invoke sub-protocols to encapsulate and re-use interaction patterns. For example, a “negotiation” protocol may use an “offer” protocol. To use a protocol, each protocol role in the “using” protocol must specify the protocols it is initiating by using a SubProtocol.

For each sub-protocol to be used, specify a SubProtocol with the “uses” as the protocol being used.

Specifying the SubProtocol maps the protocol role as follows;

?? The initiator of the SubProtocol uses the ProtocolRole of the “uses” protocol with “initiator” true.

?? The reponder of the SubProtocol uses the ProtocolRole of the “uses” protocol with initiator false.

Elements

Uses

The protocol role being used by the protocol role owning the SubProtocol.

UML

A CCA SubProtocol is modeled in UML through a Foundation::Core::Generalization, with the parent being the initiator protocol, and the child the used Protocol. See details in section 5 "UML Profile Specification", subsection " About Protocol, Port and Component (re) Use", heading " Protocol and SubProtocol ".

3.5.2.6 Request Reply Protocol

Extends

Protocol

Owned By

Package

Semantics

A very common interaction is “request/reply”. A Request Reply Protocol is a specialization of protocol to make specifying this pattern easier. The Request Reply Protocol makes the following constraints on a protocol:

?? There will be only messages, no SubProtocols.

?? There will be one initial message, all other messages will be replies to that message.

?? Only one of the replies will actually be used for any instance of the initiating message.

?? The Choreography is fixed to the initial message transitioning to the responding messages, this Choreography can not be re-specified.

Request Reply Protocol is patterned after the ebXML “Business Transaction” and is frequently only used as a sub-protocol.

Elements

None

Constraints

See above

UML

A CCA RequestReplyProtocol is modeled in UML in the same way as a CCA Protocol. See details in section 5 "UML Profile Specification", subsection "Protocol «profile» Package", heading "RequestReplyProtocol".

3.6 Component Realization

Process components are abstract, they have no specification of implementation. A components implementation may be specifies as primitive or as a composition of other components. A community shows how a set of components works together for a business purpose.

3.6.1 Conceptual Meta-Model

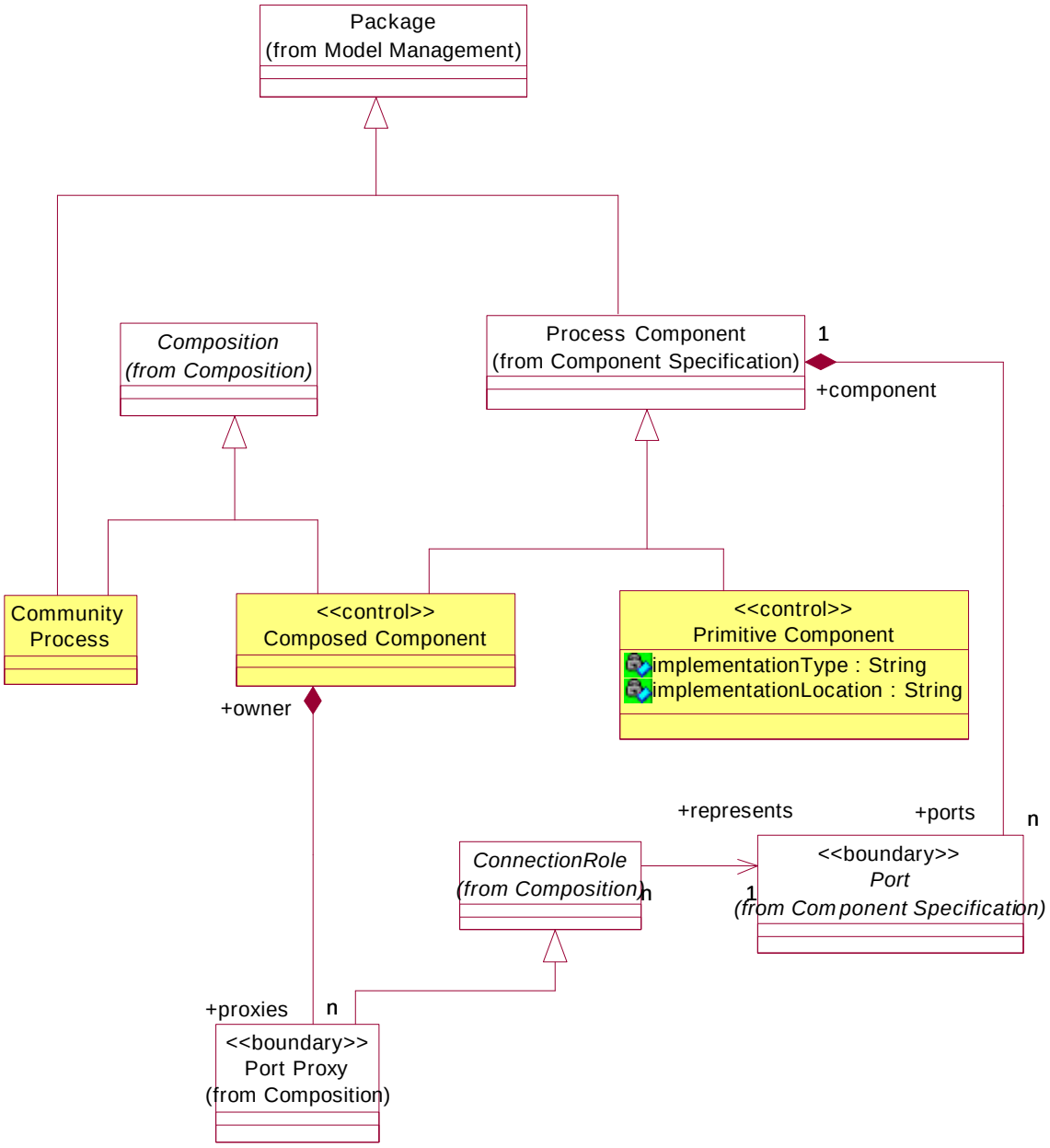


Figure 4: ComponentRealization Conceptual Meta-Model

3.6.1.1 Summary

Process components specify the abstract, external contract of the component. Such a component is realized as either a **Primitive Component** or **Composed Component**. A

primitive component is defined in a model or language outside of the scope of CCA. A **Composed Component** is defined by a CCA composition.

A composition may also be used to define a **Community Process**, which shows how members of a community collaborate within a process.

Abstract and Concrete components

This model of abstract and concrete type works exactly like abstract Vs. concrete classes in C++ & Java. An abstract class is incomplete while a concrete class is fully defined. A concrete component is "real" and can be asked to do work. Note that there is NO WAY do define the "inside" of a "Process Component", so it must be abstract & "open". "Abstract" process components can be created - these have no "insides" specified. A primitive component is not abstract in that it assumes that its insides are defined elsewhere (I.E. in Java) - but still defined. A composite component has its insides defined by composition. And, for any particular concrete composed component there is exactly one such composition.

We want to be able to have alternatives defined for any abstract component. Alternatives come via a choice of a concrete component - compositions & primitive components are alternatives and could be alternatives to the same abstract component. Inheritance supports this exactly the way it does in Java or C++. So "alternatives" are defined by making alternative components that satisfy the same contract & are therefor subtypes of the abstract "contract" of the process component. These alternative components may use composition or may be primitives and there can be any number of them defined at any time. Since some components have only one reasonable realization, it is possible to define the "inside" and "outside" in "one shot" using a primitive or composed component.

Example:

?? ProcessComponent: "CalcualePrice" defines ports and choreography.

?? ComposedComponent: CaluatePriceUsingOtherComponents1 is subtype of "CalculatePrice"

?? ComposedComponent: CaluatePriceUsingOtherComponents2 is subtype of "CalculatePrice"

?? PrimitiveComponent" CalcualtePriceUsingExternalProgram is subtype of "CalculatePrice"

?? ComposedComponent: OrderProcessor Uses "CalcualtePrice" for a "ComponentUsage" called "priceIt"

?? At runtime a trader is called and binds one of the "CalculatePriceUsing..." components to "priceIt". It knows that this is valid because the "CalculatePriceUsing..." component is a subtype of (substitutable for) CalcualePrice

3.6.2 Model Elements

3.6.2.1 Primitive Component

Extends

Process Component

Owned By

Package

Semantics

A primitive component specifies a concrete component implemented using capabilities outside of the scope of CCA – A wrapped legacy application, Java, C++ Etc.

Primitive component inherits from Process Component, allowing primitive components to define their own "contract" or to inherit a contract from an abstract process component.

Attributes are provided for the type and location of the external implementation, but CCA places no restrictions or specific semantics on these attributes. A particular implementation technology may use them as required.

Elements

ImplementationType

An attribute that is intended to be meaningful to the implementation mapping of CCA to specify the kind of primitive component, E.G. "Java" or "COM".

ImplementationLocation

An attribute that is intended to be meaningful to the implementation mapping of CCA to specify how to locate a primitive component's implementation artifacts – such as a class file or DLL.

UML

A CCA PrimitiveComponent is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem. See details in section 5 "UML Profile Specification", subsection "ComponentRealization «profile» Package", heading "PrimitiveComponent".

3.6.2.2 Composed Component

Extends

Process Component and Composition

Owned By

Package

Semantics

A composed component specifies a concrete component using a composition of other components. The composed component gives CCA its recursive component assembly capability.

The composition that is part of the composed component allows the *use of* other components to be "placed inside" of the composed component, configured and then "wired together". This is intended to support visual tools and drag-and-drop component assembly.

The "inside" of the component can be thought of as a template for a set of component *instances*. These instances serve to implement the component *type* being defined.

The ports on the components "inside" the composition will each expose usage of their ports. These ports are what can be wired together. For each port on the "outside" of the component being defined a proxy is created on the "inside" that allows the component ports on the inside to be wired to these external proxies.

In some cases a composition may use abstract Process Component's inside of a composition. Obviously such a composition is not fully concrete. By the time such a "partially abstract" composition is used, the abstract process components must be substituted with concrete components. This may be dome at design time (using contextual binding) or at runtime (using implementation specific techniques).

The semantics of composition are defined in the "Composition" package.

Elements

Proxies

For each port on the process component a Port Proxy is created (preferably by the design tool) for use in the composition. These proxies are used make connections to the "inside" of these ports.

A port may be seen as extending though the components boundary. On the outside, external components may connect to the port. On the inside, components are connected to the proxy for this external port.

Proxies have the inverse interface from the external port. That is, if a ports "sends" a document its proxies will receive that document.

UML

A CCA ComposedComponent is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem. See details in section 5 "UML Profile Specification", subsection "ComponentRealization «profile» Package", heading "ComposedComponent".

3.6.2.3 Community Process

Extends

Composition and Package

Owned By

Package

Semantics

Community processes may be thought of as the “top level composition” in a CCA specification, it is a specification of a composition of process components that work together for some purpose other than specifying another component.

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 being done “top down” or as an assembly of existing components (bottom up). When design is being done top down, it is usually the community process which comes first and then components specified to fill the roles of that process.

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

Elements

None

UML

A CCA CommunityProcess is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem. See details in section 5 "UML Profile Specification", subsection "ComponentRealization «profile» Package", heading "CommunityProcess".

3.7 Composition

Composition is an abstract capability that is used for composite components and for community processes. Compositions shows how a set of components can be used for some purpose.

3.7.1 Conceptual Meta-Model

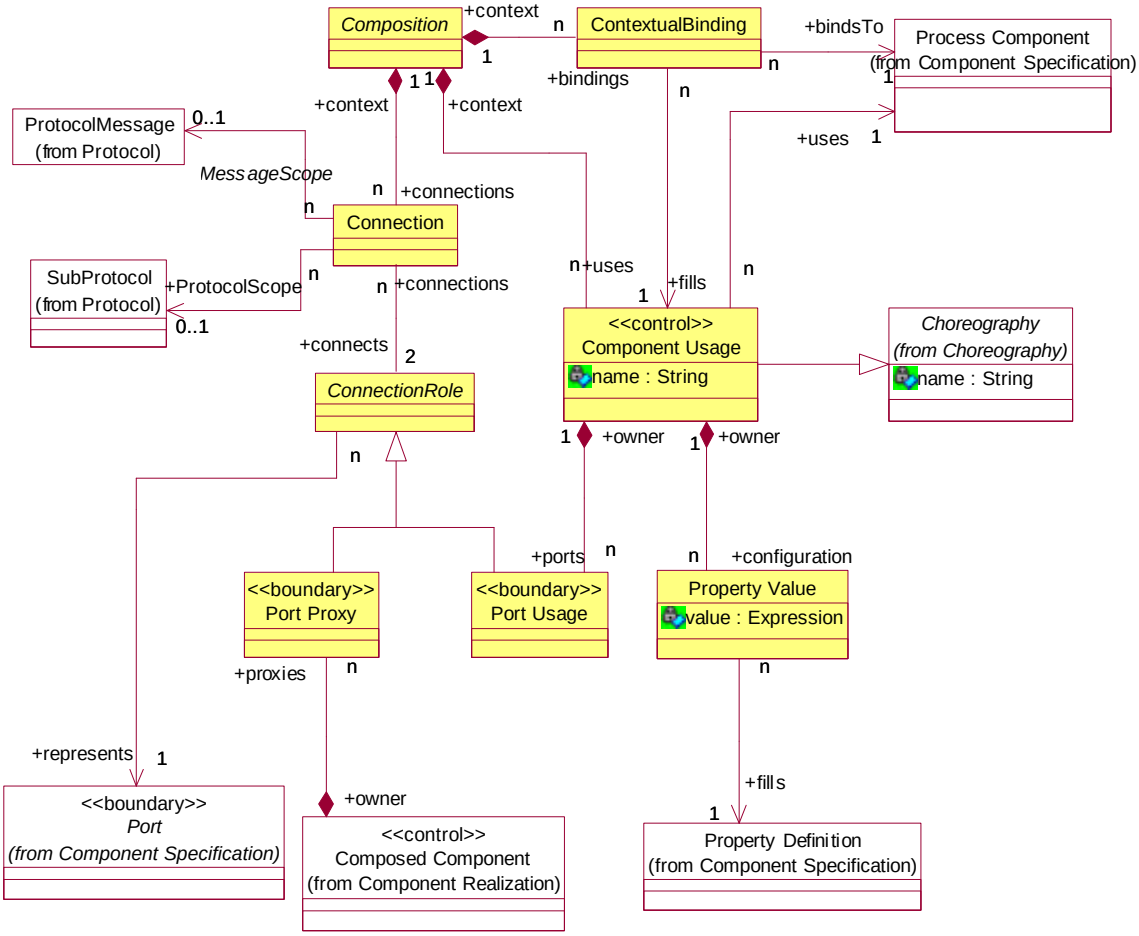


Figure 5: Composition Conceptual Meta-Model

3.7.1.1 Summary

A **composition** defines how process components work together to achieve some purpose. That purpose is either to realize a larger process component or describe a community process.

A composition contains **component usage**s to show how other process components may be used within the composition. Note that the same process component may be used multiple times for different purposes. Each time a process component is used, each of its ports will also be used with a "**Port Usage**". A port usage shows the connection point for each use of that component within the composition. The components used may be concrete (primitive of composite) or abstract (process component). If the components used are abstract, a concrete component must be bound to the usage at some later time (see ContextualBinding).

Attached to a component usage is one or more **Property Values**, configuring the component with properties that have been defined in property definitions.

A composition also contains a set of "**connections**". A connection connects compatible ports on components together. Anything sent out of one side will be received by the other side. So a connection is a form of event registration.

A connection may also connect to a **Port Proxy**. A port proxy is used when the composition realizes a process component and provides a connection point for the external ports of the process component being defined. Each port proxy represents a wiring point on the "inside" for a port on the "outside" of the component being composed.

A connection may connect a port that implements only parts of a particular protocol, such as a flow port being connected to one message in a protocol. This enables components at different levels of granularity to be connected. When this occurs the connection may have to be scoped using **Message Scope**, to select a particular message when the connection is to a flow port. Or, The connection may be constrained by a Sub-protocol using Protocol Scope.

A composition may use (**uses** relation) an abstract Process Component as well as concrete primitive or composite components. A **Contextual Binding** allows realized components to be substituted for abstract components when a composition is used. This may be done in the design or at runtime. When the substitution is done in the design a contextual binding is used. The mechanisms for runtime substitution are not defined in CCA.

When a **Choreography** is defined for a composition, it defines the sequencing of each component usage as a series of steps with transitions between these steps, forming a state machine.

3.7.2 Model Elements

3.7.2.1 Composition

Extends

Choreography (Indicating that Component Usage and Connection Roles can be choreographed).

Owned By

Package (as a Community Process or Composed Component)

Semantics

Composition is an abstract capability that is inherited by the two things that can be composed – Composed Components and Community Processes. Compositions describe how instances of process components are configured, connected and choreographed to implement the composed component or community process.

Elements

uses

The set of component usage's for the composition, this set may be considered as a template for component instances which will realize the composition.

connections

The set of connections (or wires) between port instances or port proxies. A connection registers each port as an event listener for the other, connecting the message flows between instances of components used by the composition.

bindings

The set of "ContextualBindings" for the composition, where the composition is the context for the substitution of concrete components for abstract components.

UML

A CCA Composition is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "Composition".

3.7.2.2 Component Usage

Extends

Choreographed (Indicating that process component may be choreographed by the compositions choreography).

Owned By

Composition

Semantics

A composition *uses* other components to implement the propose of the composition (a community process or composed component), "Component Usage" 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 component instances. Each component instance will have a "Component Usage" to say what kind of component it is, what its property values are and how it is connected to other components. A component usage will cause a component instance to be created at runtime.

Each use of a component will carry with it a set of "port usage" which will be the connection points to other components.

Elements

Name

The name of the usage. By default this will start with the name of the “uses” process component with some suffix to make it unique.

context

The composition which owns the component usage.

Uses

The process component to be used (which includes the subtypes of process component: primitive and composed component).

Ports

The port usage’s – one for each port on the “uses” process component. These should be created automatically by the design tool.

Configuration

Property values to configure the component based on its property definitions. Each value will set a value of the component instance created to implement the composition.

Constraints

If “uses” is an abstract Process component or a composed component using abstract process components a concrete component must be bound to the component usage prior to execution.

There must be a port usage for each port defined on the process component.

UML

A CCA ComponentUsage is modeled in UML as a Stereotype, with the same name, of Model Management::Subsystem, and a Stereotype of Foundation::Core::Class named "PropertyHolder". See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "ComponentUsage", and subsection "Common «profile» Package", heading "PropertyHolder".

3.7.2.3 Property Value

Extends

None

Owned By

Component Usage

Semantics

To be useful in a variety of conditions, component may have configuration properties – which are defined by a "property Definition". When the component is used in a "Component Usage" those properties may be set using a "Property Value". These values will be used to construct a component instance.

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

Elements

Owner

The component usage to which the property value applies.

Fills

The property definition for the value.

value

An expression returning the property value. Property expressions may only reference constant values and properties of other components.

Constraints

The type returned by the property value expression must be compatible with the type defined by the property definition.

The property value must fill a property definition of the component being used.

UML

A CCA PropertyValue is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Attribute. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "PropoertValue".

3.7.2.4 Port Usage

Extends

ConnectionRole

Owned By

Component Usage

Semantics

For each component usage there will be exactly one "Port Usage" for each port defined for the component being used. These will normally be created by the design tool.

The Port Usage provides a "connection point" for components within the composition and expose the realized protocols or data flows within the composition.

The "process Component" / "Port" pattern which defines the components external interface is essentially replicated in the "Component Usage" / "Port Usage" part of the composition. Each time a component is used, each of its ports is used as well.

Elements

Owner

The component usage for which this is a port usage.

Constraints

For each component usage there will be exactly one "Port Usage" for each port defined for the component being used.

UML

A CCA PortUsage is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "PortUsage".

3.7.2.5 Port Proxy

Extends

Connection Role

Owned By

Composed Component

Semantics

When a composition is being used to define the insides of a composed component there must be a way to connect to the ports on the "outside" of the component being defined. Port Proxy provides this capability by making a "Connection Role" within the composition for connecting to these external ports. Port proxies should be created automatically by the design tool.

Ports can be thought of as extending through the component being defined with an "external" and an "internal" connection point. The port proxy is this internal wiring point. As such it has a protocol which is the inverse of the "external" ports protocol. If the external ports sends a message, the port proxy will receive that message and forward it on to the internal components connect to the port proxy.

Elements

Owner

The composed component being defined and owning the port being represented (see "represents" in "ConnectionRole").

Constraints

For each composed component there shall be exactly one port proxy for each port defined on the composed component.

If the port proxy represents a flow port, the proxy shall have the inverse direction of the flow port.

If the port proxy represents a protocol port, the protocol role of the port proxy shall be the inverse protocol role of the represented port.

UML

A CCA PortProxy is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "PortProxy".

3.7.2.6 Connection Role

Extends

None

Owned By

Ownership is managed by concrete subtypes: Port Usage and Port Proxy.

Semantics

ConnectionRole is an abstract element which represents something that can be connected within a composition. This will either be a "Port Usage" or "Port Proxy". In either case the connection role will reference a "port" that is the basis for the connection point.

A ConnectionRoles may be an event consumer, event producer or both and may be connected by any number of connections. Each connection registers instances of the underlying port as event producer and/or consumer of the other, thus forwarding the messages between components instances.

Elements

represents

The port which the connection role represents. The connection role is bound by the constraints of the associated port.

connections

The connections attached to (or using) the connection role.

UML

A CCA ConnectionRole is abstract. Only the concrete specializations of ConnectionRole correspond to UML Stereotypes.

3.7.2.7 Connection

Extends

None

Owned By

Composition

Semantics

A connection connects the instances of two ports within a composition. Each port can produce and/or consume message events. The connection registers each port instance 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 an

instance of a component will be used and thus how it will be connected to other components *within that context*.

A connection connects exactly two Connection Roles (Port usage or Port Proxy). If connecting to Port Usage, it will be connecting to the use of a component within a composition. If connecting to a Port Proxy it will be connecting to the ports on the "outside" of the component being composed.

A connection may be thought of as a cable between two plugs. The plugs are the ConnectionRoles and the connection the cable.

Since a connection may connect a complex protocol to a simpler one or even a flow port, it may be necessary to scope the connection. Setting "ProtocolScope" to a specific Sub Protocol selects a part of that protocol. Setting MessageScope to a particular message scopes the connection to only connect that message. Setting these relations is only required when connecting ports of different granularities. In many cases tools may be able to set these based on the type of the two ports.

Elements

Context

The composition which owns the connection. Note that the connection is not owned by either of the things connected, which are ignorant of how they are used. The composition owns the component usage and how they are connected within that context.

Connects

The two ConnectionRoles (Port Proxy or Port Usage) being connected.

MessageScope

Restricts the connection to the related connection.

ProtocolScope

Restricts the connection to the related sub protocol.

Constraints

Each connection role must be owned by the same composition as the connection.

UML

A CCA Connection is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Association. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "Connection".

3.7.2.8 Contextual Binding

Extends

None

Owned By

Composition

Semantics

A composition is able to use abstract process components 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 component for a compatible abstract component when an abstract composed component is used. So within the composition that uses the abstract composed 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 specialization of CCA may subtype ContextualBinding and apply selection formula to the binding, as is common in workflow systems.

Elements

Context

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

Fills

The use of a component in which the substitution to a concrete component should take place. This component usage does not have to be within the same composition as the contextual binding, it may be anywhere the component usage occurs within the scope of the composition owning the binding.

BindsTo

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

Constraints

The process component related to by “bindsTo” must be a subtype of the component used by the component usage related to by “fills”.

UML

A CCA ContextualBinding is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Binding. See details in section 5 "UML Profile Specification", subsection "Composition «profile» Package", heading "ContextualBinding".

3.8 Choreography

Choreography allows the ordering of various actions in a system to be specified as a set of steps in a process and transitions between these steps. The base model of Choreography is that of an activity graph .

3.8.1 Conceptual Meta-Model

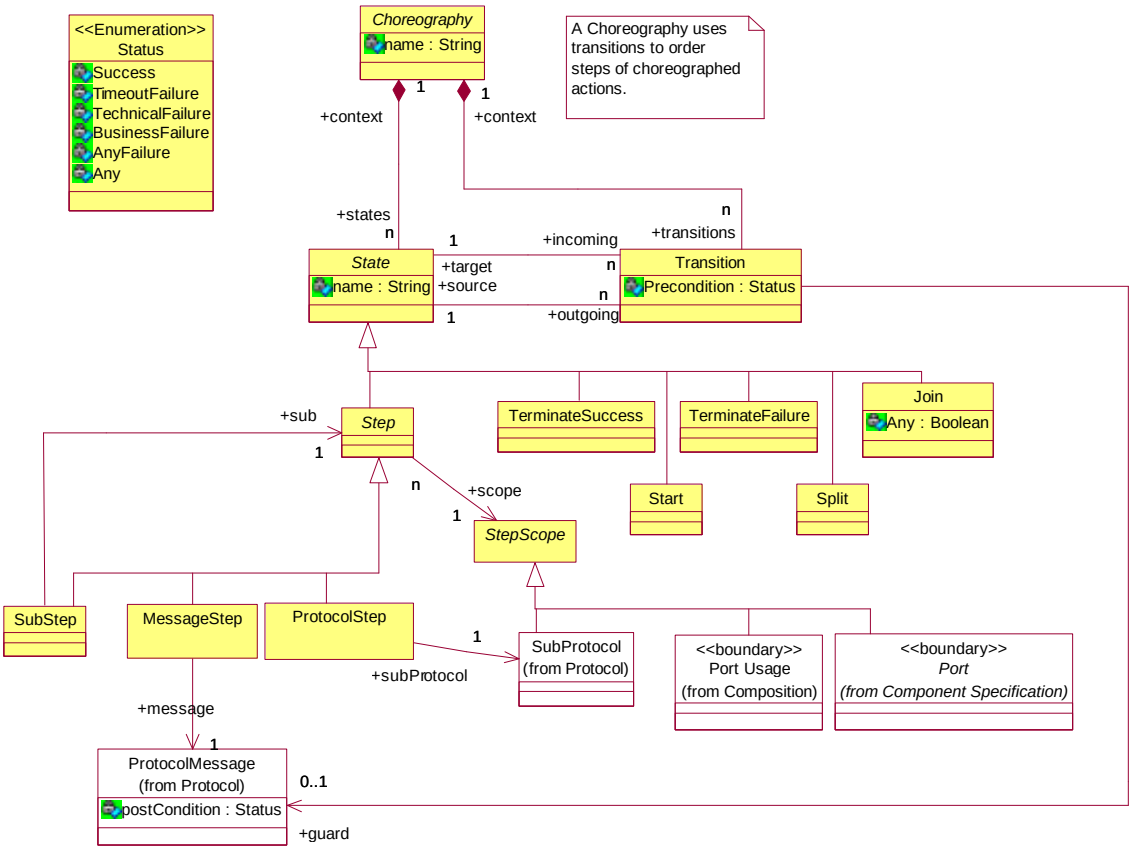


Figure 6: Choreography Conceptual Meta-Model

3.8.1.1 Summary

A **Choreography** uses transitions to order steps of choreographed actions, as a state machine. Each step in the choreography must refer to a Message or a SubProtocol. Messages and SubProtocols take on or begin some kind of action or activity within the context of the choreography.

Choreography is an abstract capability that is inherited by things that can be choreographed, such as: Process Componentsand Protocols.

Within any choreography there must be some place to **start** and places to end, either with **a Terminate Success** or a **Terminate Failure**. Concurrent steps are defined by using a **split** with transitions to each concurrent step and a **join** when the concurrent steps come back together.

The ordering of steps is controlled by **transitions** between states (step being a kind of state). Transitions specify flow of control that will occur if the conditions (Precondition and Guard) are met.

A **Sub Action** allows choreography of interactions within protocols where interactions are also defined as steps.

Each action will have a termination **status** of success or one of several kinds of failure.

Choreography may be used at multiple levels;

?? A protocol Choreography specifies the sequencing of messages and sub-protocols between protocol roles. This is much like a sequence diagram.

?? A process component Choreography specifies the sequencing of multiple messages and sub-protocols of ports and is part of the external contract of the component.

The use of choreography at all of these levels is not always required, as sufficient specification may be determined from the other layers.

3.8.2 Model Elements

3.8.2.1 Choreography

Extends

None (Abstract Capability)

Owned By

Ownership is based on concrete model element which inherits from Choreography.

Semantics

Choreography is an activity graph owning a set of states and transitions and specifying an ordering of these states based on the transitions. The states that perform actions are "Steps" of the process being choreographed.

Elements

States

The set of states being choreographed and, indirectly (through "step") the set of actions being choreographed.

Transitions

The transitions which order the states and steps.

UML

A CCA Choreography is abstract.

CCA Choreography, is modeled in UML as an ActivityGraph, aggregated in the context of the UML stereotype for Protocol or ProcessComponent. See "UML Profile Specification", subsection "Choreography «profile» Package", heading "Choreography".

3.8.2.2 State

Extends

None

Owned By

Choreography

Semantics

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

Elements

Context

The owner of the state.

Name

The name of the state

Incomming

The set of all possible transitions into this state.

Outgoing

The set of all possible transitions out of this state.

Constraints

Incoming and outgoing transitions will be within the same choreography.

UML

A CCA State is abstract. Only the concrete specializations of State correspond to UML Stereotypes.

The UML modeler may use UML Behavioral Elements::State Machines::State for the specification of StateMachines and ActivityGraphs.

3.8.2.3 Transition

Extends

None

Owned By

Choreography

Semantics

States represent a condition of some process. Transitions represent the movement from one state to another, or a flow of control. The transitions may have conditions which control if it is or is not a legal transition in a given circumstance, this is expressed using the PostCondition and Guard.

If there are multiple legal transitions out of a state, it is up to the implementation of that state to pick the actual transition from the set of potential transitions..

Elements

Context

The choreography owning the transition.

Source

The state from which the transition occurs.

Target

The state to which the transition occurs.

PreCondition

The termination status of the prior state which must be true for the transition to take place (be legal). Default: Any

Guard

The termination action of the prior step which must have happened for the transition to take place (be legal).

UML

A CCA Transition is modeled in UML as a Stereotype named "ChoreographyTransition", of Behavioral Elements::State Machines::Transition. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "ChoreographyTransition".

3.8.2.4 Step

Extends

State

Owned By

Choreography

Semantics

A step is a state in a choreographed process that does real work by performing some action.

There are three kinds of Steps : MessageStep, ProtocolStep and SubStep.

Elements

scope

The Port, PortUsage or SubProtocol that defines the context for the Step.

UML

A CCA Step is abstract and not directly modeled in UML. Its specializations are modeled in UML as specified below.

3.8.2.5 MessageStep

Extends

step

Owned By

Choreography

Semantics

A MessageStep is a Step for sending or receiving a ProtocolMessage.

Elements

message

The ProtocolMessage to be sent or received.

UML

A CCA MessageStep is modeled in UML as a Stereotype, named "MessageStep" of Behavioral Elements::State Machines::Transition. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "MessageStep".

3.8.2.6 ProtocolStep

Extends

State

Owned By

Choreography

Semantics

A SubProtocolStep is a step for launching the activity of a whole SubProtocol .

Elements

subProtocol

The SubProtocol to activate in the ProtocolStep.

UML

A CCA ProtocolStep is modeled in UML as a Stereotype, named "ProtocolStep" of "Step", of Behavioral Elements::Activity Graph::ActionState. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", headings "ProtocolStep".

3.8.2.7 SubStep

Extends

State

Owned By

Choreography

Semantics

A SubStep is a step in a choreographed process, inserted to reference and drill down into an specific Port, PortUsage or SubProtocol, such that inner MessageStep or SubProtocolStep unequivocaly refer to the desired Message or SubProtocol .

A SubStep is used within the context of another step, such as a message within a protocol. SubStep enables the choreography of fine-grain actions.

Elements

sub

The nested Step to execute within the scope of the SubStep.

UML

A CCA SubStep is modeled in UML as a Stereotype, named "SubStep" of Behavioral Elements::Activity Graph::SubactivityState. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", headings "SubStep".

3.8.2.8 Start

Extends

State

Owned By

Choreography

Semantics

Start is an implicitly created state that represents a choreographed element that is ready to start and will start based on the transitions from the start state.

Elements

None

UML

A CCA Start is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::State Machines::Pseudostate, of kind #initial. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "Start".

3.8.2.9 TerminateSuccess

Extends

State

Owned By

Choreography

Semantics

The TerminateSuccess state is an implicitly generated state that is the normal, successful completion of a choreography. When TerminateSuccess is reached the action of the choreographed element is done.

Elements

None

UML

A CCA TerminateSuccess is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::State Machines::FinalState. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "TerminateSuccess".

3.8.2.10 TerminateFailure

Extends

State

Owned By

Choreography

Semantics

The TerminateFailure state is an implicitly generated state that is transitioned to when the choreographed element ends in failure. When TerminateFailure is reached the action of the choreographed element is done. In a business sense, failure is indicated when the business intent of the choreography was not satisfied. E.G. when an order was not accepted.

Elements

None

UML

A CCA TerminateFailure is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::State Machines::FinalState. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "TerminateFailure".

3.8.2.11 Split

Extends

State

Owned By

Choreography

Semantics

A split is used to indicate that all legal transitions from the split state will occur. It is undefined if these will happen concurrently or in parallel. This may be distinguished from any other step in which only one transition from any state may occur.

Elements

None

UML

A CCA Split is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::State Machines::Pseudostate, of kind #fork. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "Split".

3.8.2.12 Join

Extends

State

Owned By

Choreography

Semantics

A join is used to combine actions that had been split. If “any” is true, the first transition to the join will conclude the spit and all other actions of the split will be terminated. If any is false, all actions of the split must conclude for the join to be satisfied and transition out.

Elements

Any

True if the first transition to the join terminates the join. (Default: false)

UML

A CCA Join is modeled in UML as a Stereotype, with the same name, of Behavioral Elements::State Machines::Pseudostate, of kind #join. See details in section 5 "UML Profile Specification", subsection "Choreography «profile» Package", heading "Join".

3.9 Document Model

The document model defines the information that can be transferred between and manipulated by process components.

3.9.1 Conceptual Meta-Model

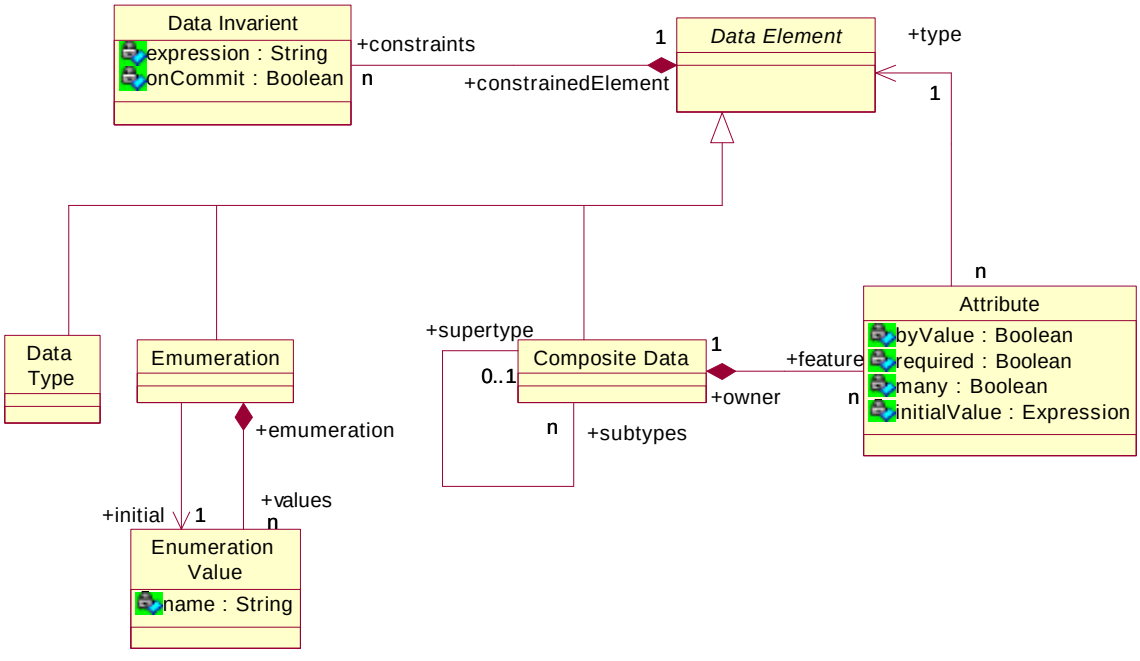


Figure 7: DocumentModel Conceptual Meta-Model

3.9.1.1 Summary

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

Attributes may be **byValue**, 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. **Primitive data** types define anything from integers to movies – these types are defined outside of CCA. An **enumeration** defines a type with a fixed set of values

3.9.2 Model Elements

3.9.2.1 Data Element

Extends

None

Owned By

Package

Semantics

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

Elements

Constraints

The set of rules that are applied to the data type.

UML

A CCA DataElement is abstract. Only some of the concrete specializations of DataElement correspond to UML Stereotypes.

3.9.2.2 Data Type

Extends

Data Element

Owned By

Package

Semantics

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

Primitive data types have their structure and semantics defined outside of CCA.

Elements

none

UML

Corresponds to standard and User Defined UML DataTypes.

3.9.2.3 Enumeration

Extends

Data Element

Owned By

Package

Semantics

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

Elements

Values

The set of values the enumeration may have.

Initial

The initial, or default, value of the enumeration.

Constraints

The names of all enumeration values must be unique within the enumeration.

UML

Corresponds to User defined enumeration stereotypes of UML DataType.

3.9.2.4 Enumeration Value

Extends

None

Owned By

Enumeration

Semantics

A possible value of an enumeration.

UML

The values of User defined enumeration stereotypes of UML DataType.

Elements

Enumeration

The owning enumeration.

Constraints

3.9.2.5 Composite Data

Extends

Data Element

Owned By

Package

Semantics

A data type composed of other types in the form of attributes.

Elements

Feature

The attributes which form the composite.

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

The names of all attributes must be unique within the scope of the composite.

UML

A CCA CompositeData is modeled in UML as a Stereotype, with the same name, of Foundation::Core::Class. See details in section 5 "UML Profile Specification", subsection "DocumentModel «profile» Package", heading "CompositeData".

3.9.2.6 Attribute

Extends

None

Owned By

Composite Data

Semantics

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

Elements

Owner

The composite of which this is an attribute.

Type

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

ByValue

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

Required

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

Many

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

InitialValue

An expression returning the initial value of the attribtue.

UML

A CCA Attribute corresponds to the UML model element of same name.

3.9.2.7 Data Invariant

Extends

None

Owned By

Package

Semantics

A constraint on the legal values of a data element.

Elements

ConstrainedElement

The data element that will be constrained.

Expression

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

OnCommit (Default: False)

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

UML

A CCA DataInvariant corresponds to a UML Foundation::Core::Constraint.

3.10 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.

3.10.1 Conceptual Meta-Model

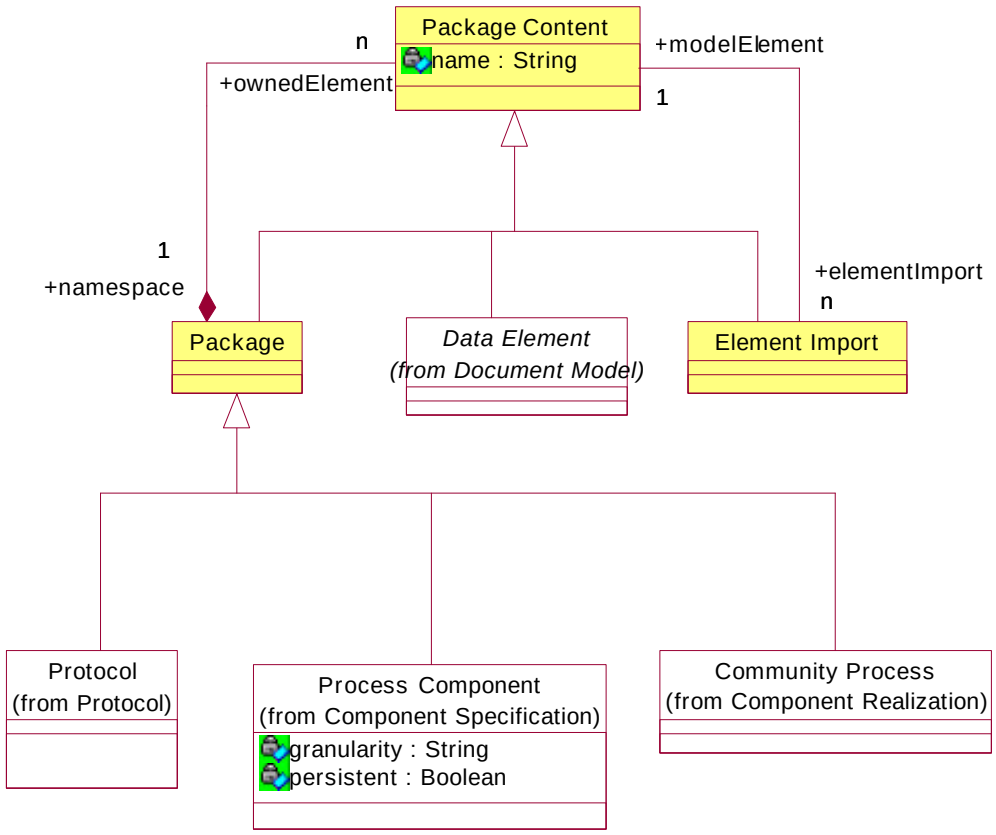


Figure 8: ModelManagement Conceptual Meta-Model

3.10.1.1 Summary

A **package** defines a logical hierarchy of reusable model elements. Elements that may be defined in a package are **Package Content** and may be Process Components, Protocols, Data Elements, Community Processes and other packages. A **Imported Element** defines a 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 process components are also packages, allowing elements which are specific to that component to be defined within the scope of that component.

3.10.2 Model Elements

3.10.2.1 Package

Extends

None

Owned By

Package or global scope

Semantics

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

Elements

OwnedElements

The content of the package.

UML

A CCA Package corresponds to the UML model element of same name.

3.10.2.2 Package Content

Extends

None (Abstract Capability)

Owned By

Package

Semantics

An abstract capability that represents an element that may be placed in a package.

Elements

Name

The unique name of the element within the package

Constraints

Names must be unique.

UML

A CCA PackageContent is abstract. Corresponds to the UML abstract ModelElement.

3.10.2.3 Element Import

Extends

None

Owned By

Package

Semantics

Defines an "Alias" for one element within another package.

Elements

ModelElement

The base element to have aliases.

UML

A CCA ElementImport corresponds to the UML model element of same name.

3.11 Combined Model Diagram

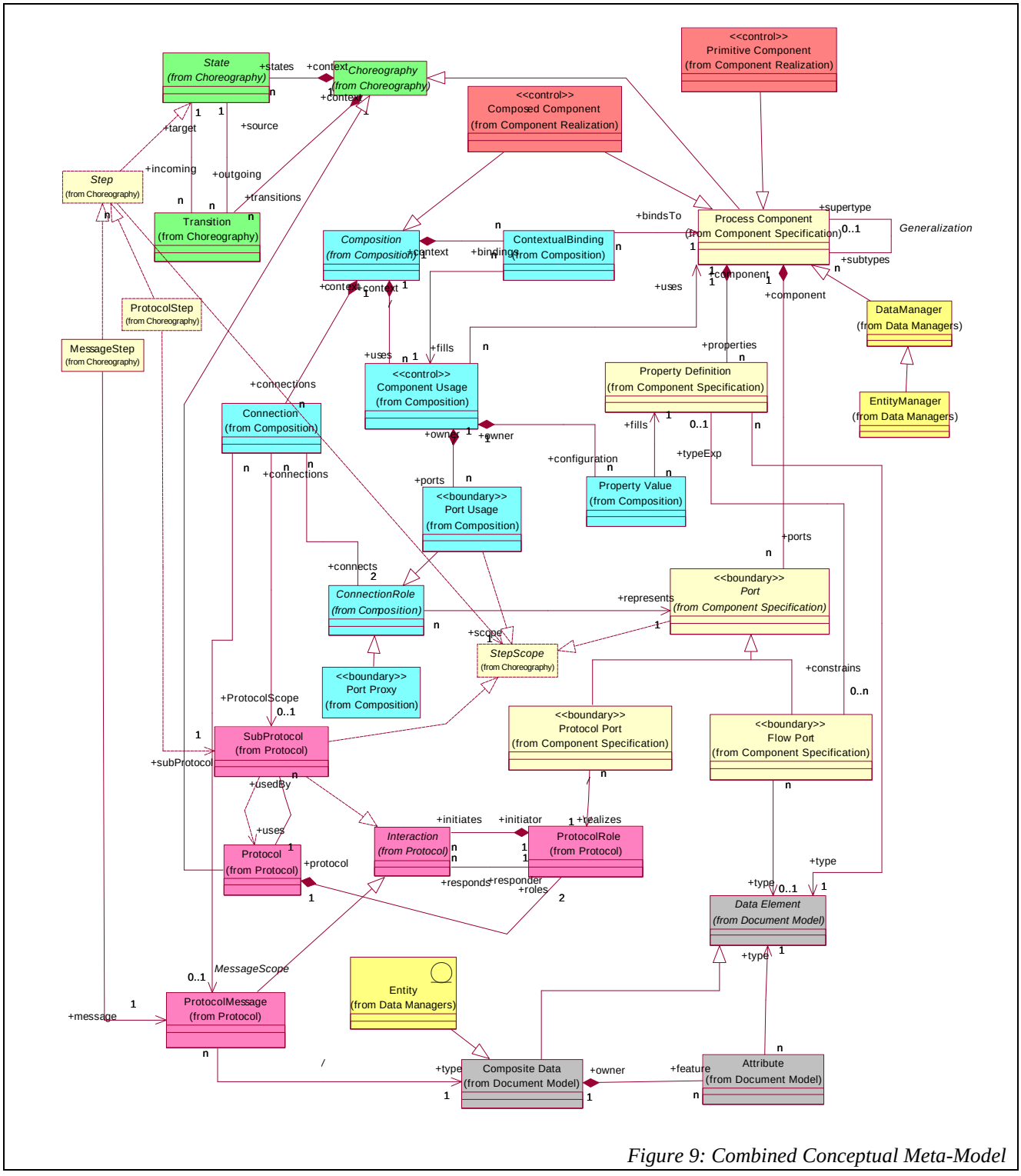


Figure 9: Combined Conceptual Meta-Model

4. 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 (Section 5) shown how CCA is expressed in the UML Meta-Model which has standard notation. The following are additions this base UML notation.

4.1 Process Component Specification Notation

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

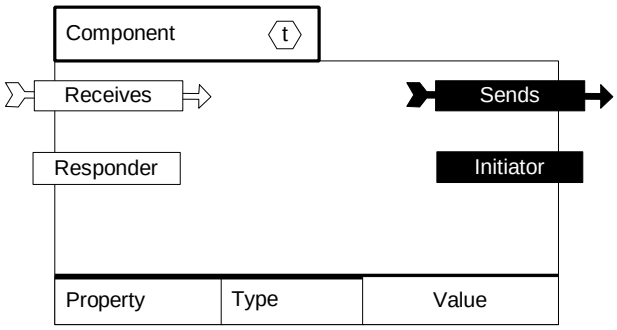


Figure 10: ProcessComponent specification notation

?? A **process component** represents its external contract as a subsystems with the following addition:

?? The process component **type** may be represented as an icon in the component name compartment. “t” above.

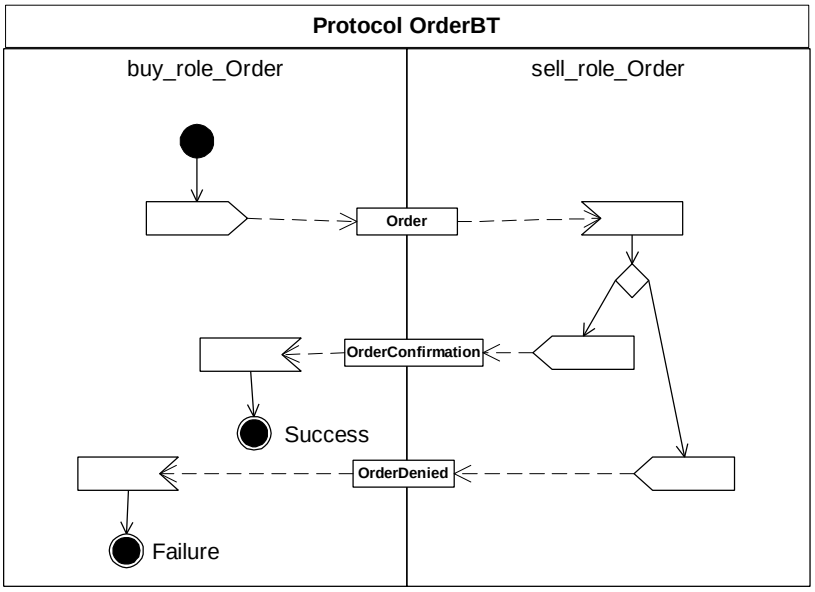
?? **Ports** are represented as going through the boundary of the box. The port is itself a smaller rectangle with the name of the port inside the rectangle.. In the above, “Receives”, “Sends”, “Responder” and “Initiator” are all ports. The type of the port is not represented in the diagram.

?? **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** 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.

?? **Property Definitions** s 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.

4.2 Protocol Notation



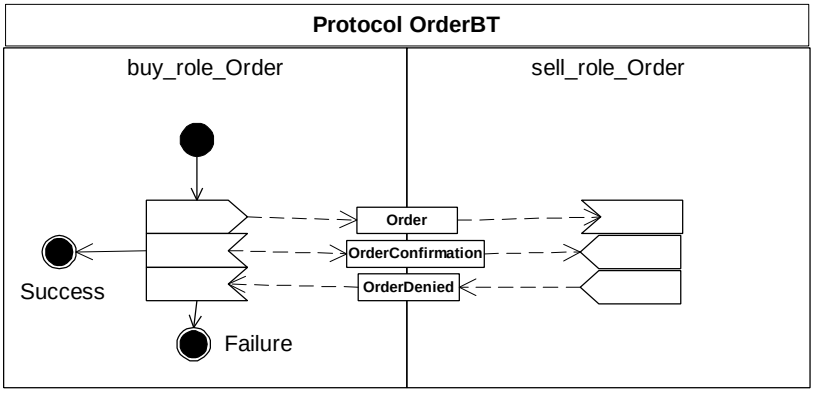


Figure 11 Protocol Notation (1)

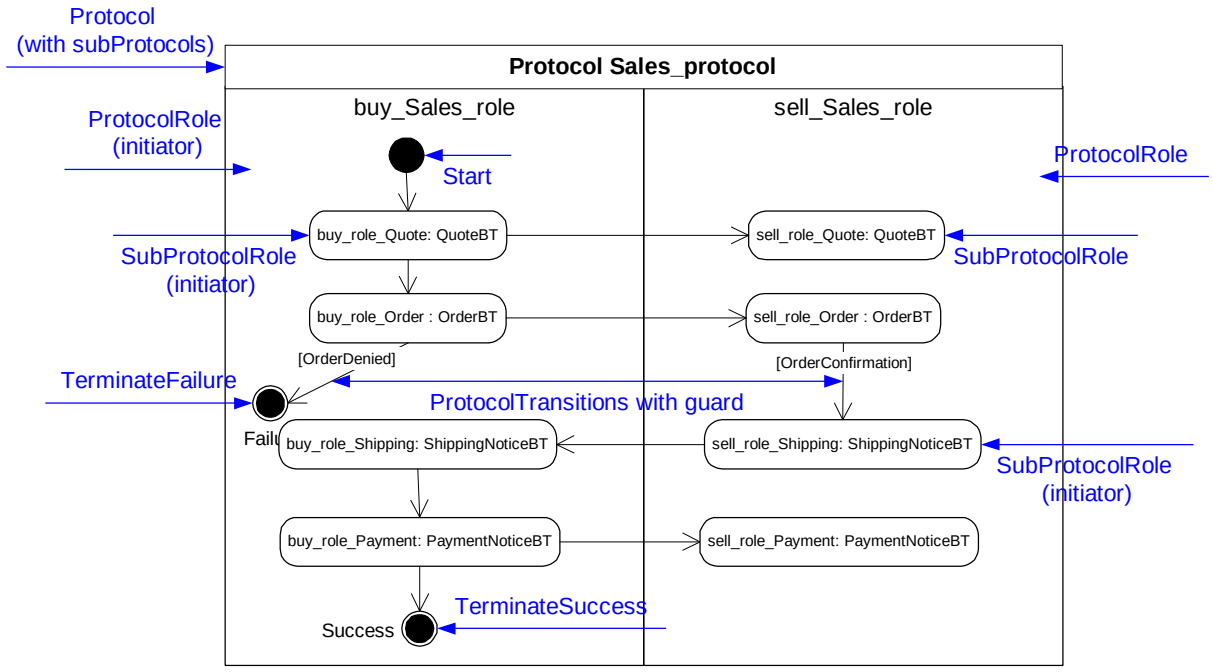


Figure 12: Protocol notation (2)

Protocols are based on UML activity diagrams, consider the following template of a protocol diagram with Choreography

A protocol uses the standard UML activity diagram notation with the following conventions;

?? The **Protocol Roles** are shown as swim lanes. The Initiator is the left most swim lane. The name of the protocol role is the heading of the swim lane.

?? The protocol is shown in terms of the initiator, using the initiator swim lane. **Start states** and **h**are shown in this swim lane.

?? A **Message** that is not a return is shown as a signal

?? A **Message return** is shown as a signal reception under the message it is a return for.

?? **Sub Protocols** are shown as action states.

?? **Sub Steps** are shown nested within the containing step.

?? The **fail state** is shown as a terminal state with the word “fail” in the center.

?? **Split** is shown as a fork

?? **Transitions** are shown as transitions.

4.3 Composite Component Notation

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

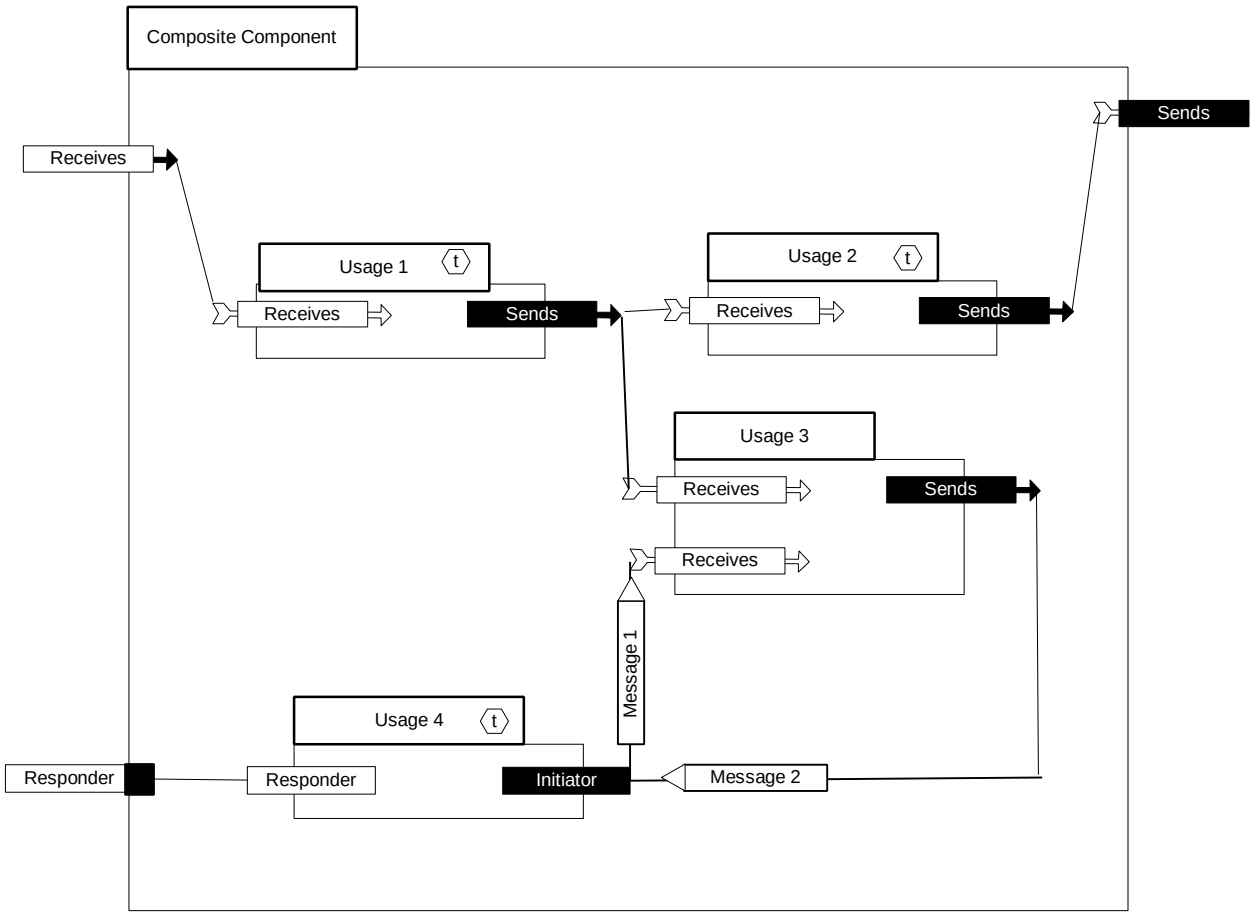


Figure 13: Composite Component notation

?? The **ports** on the composite component being defined are shown in the same way as they are on a process component.

?? The interior color of flow port arrows are inverted in color to show the **port proxy**. "Receives" and "Sends" on "Composite Component" are ports of the composite components with port proxies on the "inside".

?? The interior portion of a protocol port is inverted in color to show a **port proxy**.

?? A **component usage** is shown as a smaller version of a process component inside the composite component. Note Usage (1..4) are component usages.

?? **Port usages** are shown in the same fashion as ports, on component usages. The ports on Usage 1..4 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, or may be suppressed.

?? **Message Scope & Protocol Scope** are shown as annotations on a connection, within a box. Note that the "initiator" port on "Usage 4" is a protocol. The connectors containing Message 1 and Message 2 are being scoped to messages within the initiator's protocol so that "Usage 3" may deal with these as data flows.

4.4 Primitive Component Notation

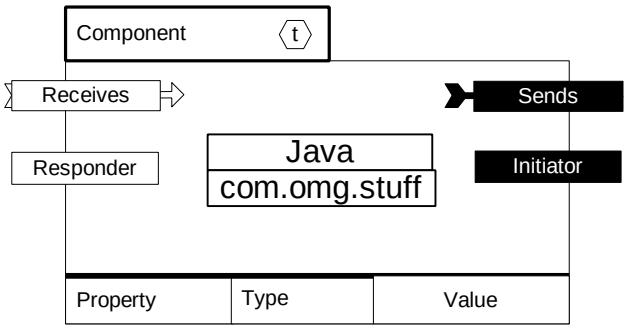


Figure 14: PrimitiveComponent notation

A primitive component is shown in the same format as a process component. The primitive component attributes are shown in the center of the central compartment.

4.5 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.

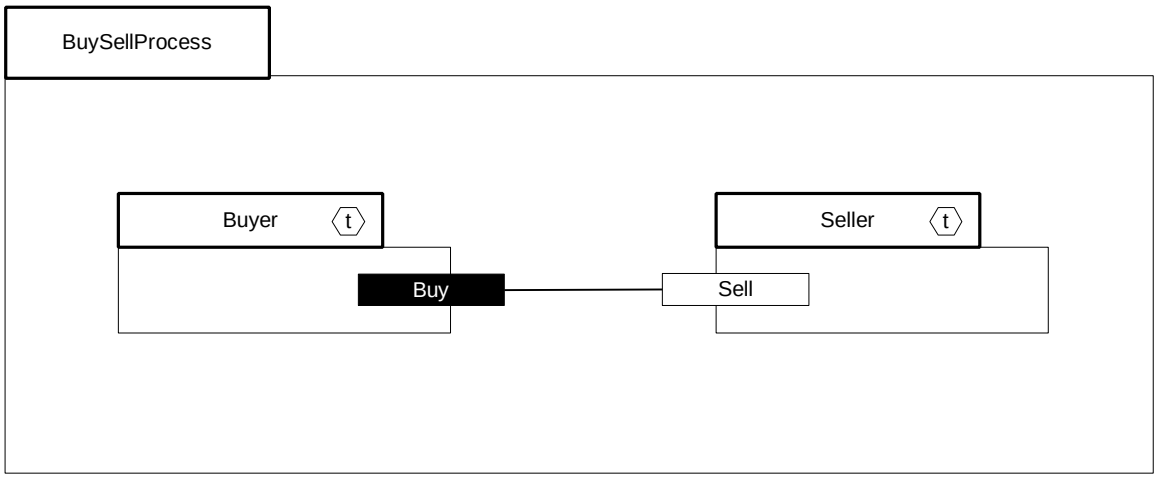


Figure 15: CommunityProcess notation

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.

4.6 Composition Notation

Being an abstract capability, composition has no specific notation. See component realization.

4.7 Choreography Notation

Choreography uses UML activity graph notation.

4.8 Data Model Notation

CCA Data Elements are in the form of a UML class with a "wavy bottom", as is the common representation of a document in a flow chart. The attributes of a composite are shown in the single compartment using standard UML notation.

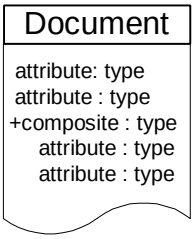


Figure 16: DataModel notation

Composite attributes may be expanded to show composite detail.

4.9 Model Management Notation

Model Management uses standard UML notation.

4.10 Data Manager Notation

The managed type is shown as a component with the managed type inside of the component.

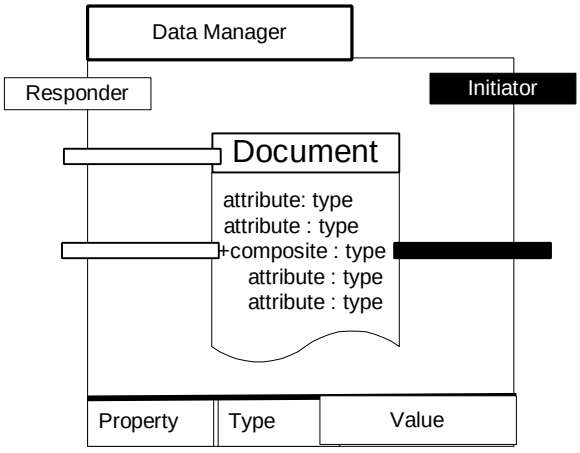


Figure 17: DataManager notation

The embedded document is managed by the data manager.

Document management ports (Black and white rectangles extending through the data manager's boundary) are not labeled as the are shown connecting to the document. Document management ports which modify the document are whie while ports that report changes in the document are black.

5. UML Profile Specification

5.1 Introduction

The UML Profile specifies how to use UML to produce specifications compliant with the Component Collaboration Architecture (CCA).

This document refers to UML as in its specification version 1.4 [UML1.4].

Reference literature about related concepts, outside of OMG standards, may be found in [OORAM], [CATALYSIS], [ROOM] and [UML-RT].

5.2 Relationship with Conceptual Meta-Model

This section specifies CCA as a UML profile, through a set of stereotypes, tagged values and constraints. The UML profile is shown in relation to the Conceptual Meta-Model for CCA, and provides the capability to support CCA by standard UML tools.

Most elements of the CCA Meta-Model directly correspond to UML elements or are logical subtypes of them. When CCA and UML metamodel elements have the same name it may be assumed that have the same semantics.

Please refer to previous sections, for a UML independent description of CCA semantics.

5.3 Choice of UML elements

The choice of UML model elements intends to facilitate the use of standard and existing UML tools to specify models with the semantic constructs of CCA.

UML Classes and Attributes are used to describe the structured data that comprises the information payload sent with messages. UML Class is stereotyped for CompositeData.

The profile uses primarily the UML Subsystem, as the unit for both classification and organization. Subsytem is stereotyped for Protocol, RequestReplyProtocol, FlowProtocol, ProcessComponent, Composition, ComponentUsage, ComposedComponent, PrimitiveComponent and CommunityProcess.

UML Class is stereotyped for ProtocolRole, ProtocolPort, RequestReplyPort, FlowPort, PortUsage and PortProxy.

UML Association is stereotyped for Connection.

UML Class is stereotyped for PropertyHolder (a necessary addition to the UML profile).

UML Attribute is stereotyped for PropertyDefinition and PropertyValue.

The behavioral element in the CCA profile is the UML Reception stereotyped as ProtocolMessage, and associated Signals.

UML Collaborations can be used to provide optional views of Protocols and Compositions.

UML ActivityGraph and StateMachine elements are used to specify the Choreography of messages and sub-Protocols in CCA, for Protocols, ProcessComponents and ComponentUsages.

UML ActivityGraph can be used to provide a high level representation on the Choreography of whole Compositions.

Standard UML Model Management artifacts, like Model and Package, can be used to organize CCA models.

A number of convenience abstract Stereotypes have been defined, to serve as common supertypes and provide containment and inheritance at the more general levels.

5.4 Profile structure

The UML Profile for the Component Collaboration Architecture is organized in the following packages :

?? Component Specification – of a collaborative party as a fully encapsulated, configurable artifact.

?? Protocol - for the specification of the set of messages that can be exchanged between collaborating parties.

?? Component Realization – specifying the realization of components as a primitive implementation, or as a composition of other components. To build a community out of components.

?? Composition – as a network of encapsulated artifacts.

?? Choreography – to specify the valid sequences of messages and activities in a set of collaborating parties

?? Document Model – that allows the specification of message payload documents.

?? Common - convenience abstract semantic supertypes.

?? Owners – convenience abstract container supertypes.

5.4.1 Packages model

The following is a model showing the Packages of the Profile, the ones used from the standard UML Meta-Model, and the dependencies between Packages.

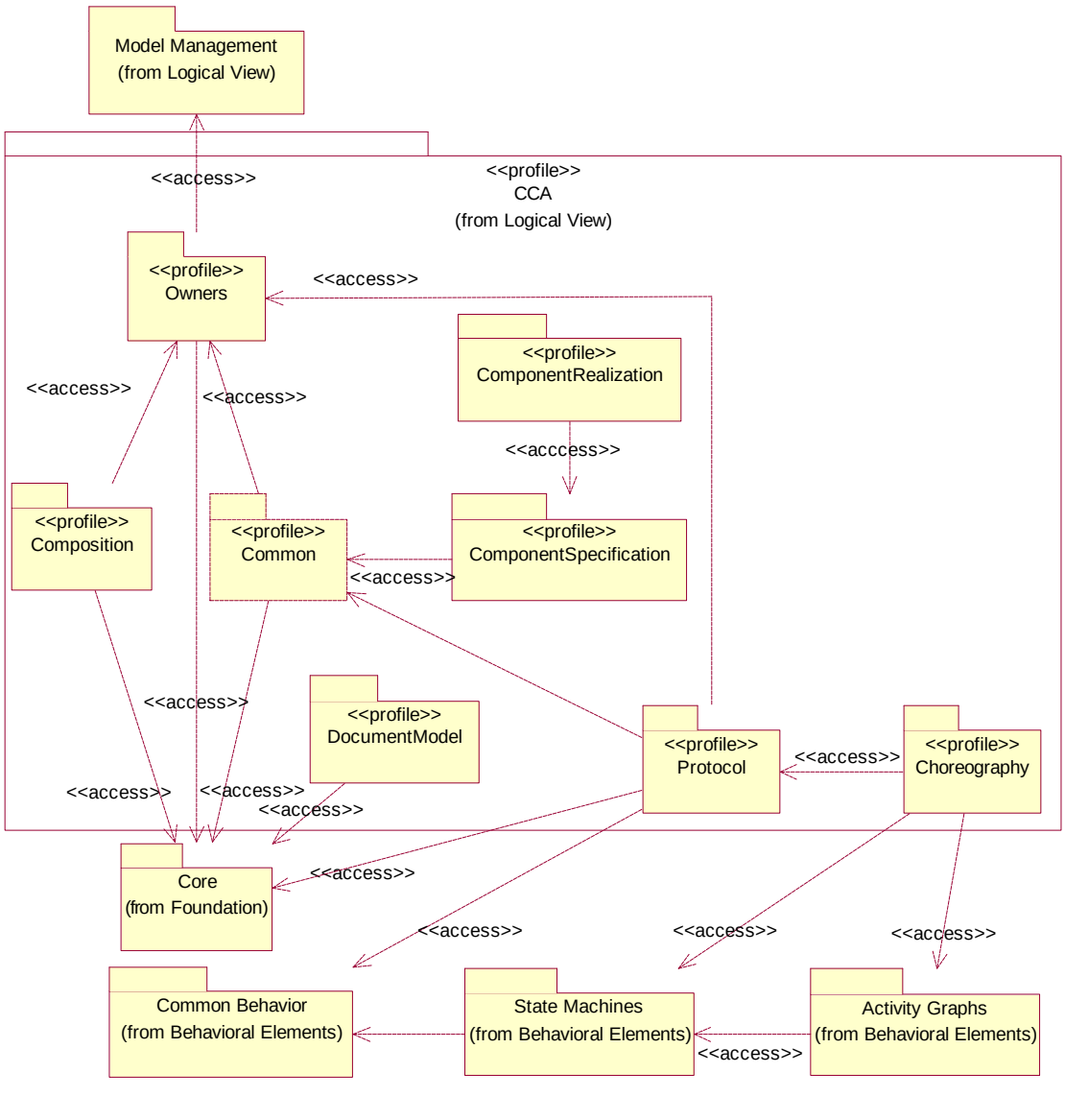


Figure 18: Structure and dependencies of the CCA «profile» Packages

5.5 ComponentSpecification «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.5.1 Virtual metamodel

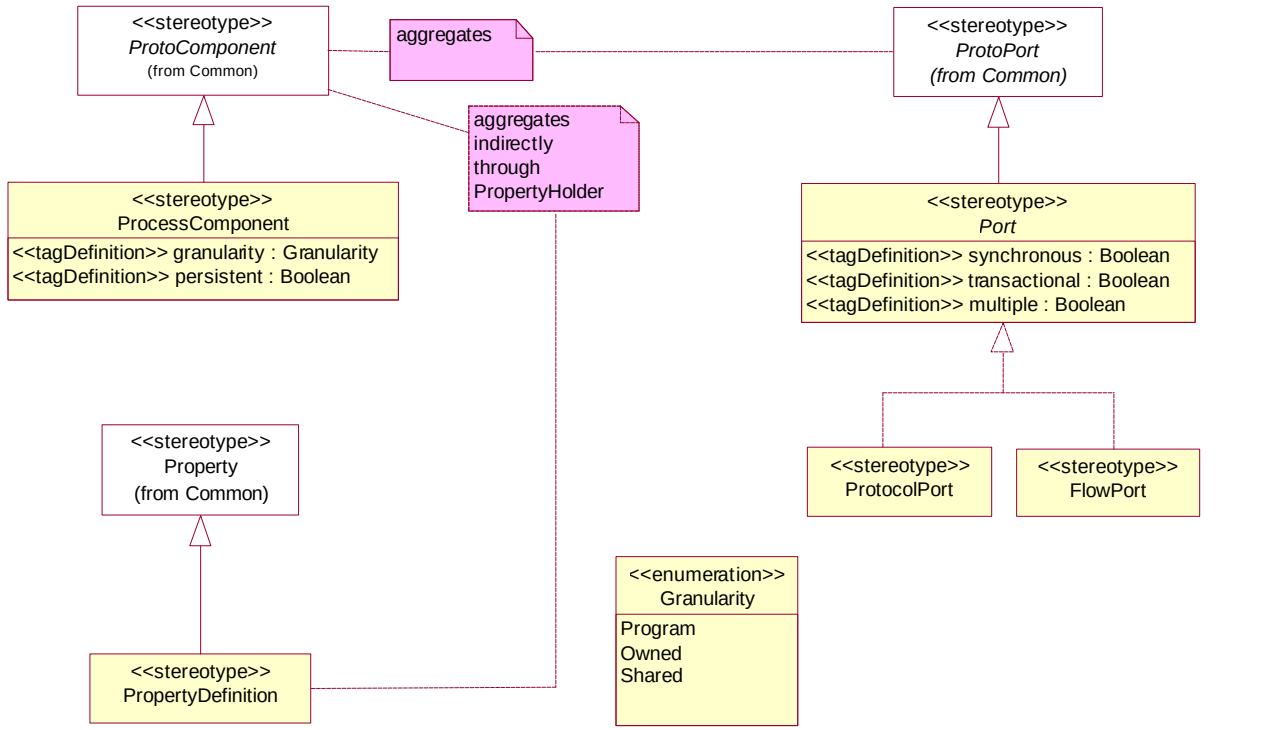


Figure 19: Class Diagram of the Virtual metamodel for ComponentSpecification «profile» Package

5.5.2 Applicable subset

From Model Management

?? Subsystem – stereotyped as ProcessComponent

From Foundation::Core

?? Class – stereotyped as Port, ProtocolPort and FlowPort

?? Attribute – stereotyped as PropertyDefinition

5.5.3 Accessed Packages

The ComponentSpecification «profile» Package accesses the Common «profile» Package.

5.5.4 Rationale

ProcessComponent is a Stereotype of Subsystem, that may contain ProtocolPort and FlowPort, as its boundary objects.

PropertyDefinition is an Attribute used for configuration of the ProcessComponent.

Because a UML Subsystem is constrained and can not contain Attributes, a Class stereotyped as PropertyHolder has to be introduced, contained in the ProcessComponent, and actually containing the PropertyDefinition.

5.5.5 «ProcessComponent»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«ProtoComponent»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Inherits from «ProtoComponent» the capabilities to own :

?? Specializations of «ProtoPort» : «ProtocolPort» and «FlowPort»

?? The utility Class of «PropertyHolder», to indirectly own «PropertyDefinition»

?? «Composition», yet only its specialization «ComposedComponent» will actually have «Composition».

Tagged Values

name = "granularity"

tagType = Granularity multiplicity = 1 tagValue= "Program"

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

name = "persistent"

tagType = Boolean multiplicity = 1 tagValue= FALSE

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

Standard UML Generalization can be used to produce more specific ProcessComponent, by specialization of a more generic one. The ProcessComponent child of the Generalization will inherit the Port of the Generalization parent ProcessComponent. The child will also inherit the PropertyHolder of the parent, and therefore its PropertyDefinition.

Constraints

In compliance to UML visibility and access rules between Packages, the ProcessComponent must have access to the Protocol containing the ProtocolRole realized by each ProtocolPort in the ProcessComponent.

For each Protocol with ProtocolRole realized by ProtocolPorts of the ProcessComponent, there must be an access Dependency with the ProcessComponent as client and the used Protocol as provider.

There is no need to define additional constraints in CCA. The constraints defined by UML already prevent the usage of ProtocolRole from ProtocolPort of ProcessComponent, if the ProcessComponent is not client of an «access» Dependency of which the Protocol is supplier.

5.5.6 «Port»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«ProtoPort»	Abstract

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Inherits from «ProtoPort» the capability to contain «ProtocolMessage»

Tagged Values

name = "synchronous"

tagType = Boolean multiplicity = 1 tagValue= "Program"

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

name = "transactional"

tagType = Boolean multiplicity = 1 tagValue= FALSE

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

name = "multiple"

tagType = Boolean multiplicity = 1 tagValue= FALSE

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

To specify the reference 'realizes' in the CCA Conceptual Meta-Model, from a ProtocolPort, to the ProtocolRole that the specifies the ProtocolMessages that may flow through the ProtocolPort, the UML Profile for CCA utilizes a standard Generalization, with the Generalization parent being the ProtocolRole, and the Generalization child the ProtocolPort. Same applies for FlowPort and FlowRole.

When using standard UML Generalization, to produce a more specific ProcessComponent, by specialization of a more generic one, a standard UML Generalization can be used to extend, in the child ProcessComponent, a Port specified in the parent ProcessComponent. The child Port of the Generalization may realize additional ProtocolRole, therefore extending the set of ProtocolMessage that may flow through the Port.

5.5.7 «ProtocolPort»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«Port»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The 'realizes' reference in the CCA Conceptual Meta-Model, from «ProtocolPort» to «ProtocolRole» is specified in the UML Profile for CCA, with a Generalization relationship with its parent being the «ProtocolRole» and its child the «ProtocolPort».

Constraints

A «ProtocolPort» realizes a «ProtocolRole»

5.5.8 «FlowPort»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«Port»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Note that in the UML Profile for CCA, «FlowPort» references directly a 'type' «DataElement», specifying the information that will be sent or received through the «FlowPort».

But in UML, every kind of port specifies its interaction capabilities by realizing a «ProtocolRole» in a «Protocol», owning «ProtocolMessage» Stereotype of Reception.

To enforce the concept of «FlowPort», additional Stereotypes named «FlowProtocol» and «FlowRole» are introduced in the UML Profile for CCA.

A «FlowProtocol» will 'realize' only «FlowRole». It is in «FlowRole» where the constrains of the CCA Conceptual Meta-Model for «FlowPort» will be effectively enforced.

The 'type' reference in the CCA Conceptual Meta-Model, from «FlowPort» to «DataElement», is substituted in the UML Profile for CCA, with a Generalization relationship with its parent being the «FlowRole» and its child the «FlowPort», the very same mechanism to specify the 'realizes' reference from «ProtocolPort» to «ProtocolRole».

Constraints

A «FlowPort» realizes a «FlowRole».

5.5.9 «PropertyDefinition»

BaseClass	Supertype	Abstract
Foundation::Core::Attribute	«Property»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The attribute 'initial' in the CCA Conceptual Meta-Model corresponds in the UML Profile for CCA, to the 'initialValue' metaattribute of Attribute.

Because constraints in UML prevent Subsystem from having StructuralFeature, ProcessComponent is not able to directly contain PropertyDefinition (an Stereotype of the Attribute StructuralFeature). To allow ProcessComponent to contain PropertyDefinition, a Stereotype of Class, named PropertyHolder (see profile Package Common in Section 5.12in page 103). PropertyHolder will contain the PropertyDefinition, providing this way a means for the ProcessComponent to contain PropertyDefinition, albeit indirectly.

When using standard UML Generalization, to produce a more specific ProcessComponent, by specialization of a more generic one, a standard UML Generalization can be used to extend or override, in the child ProcessComponent, the PropertyDefinition specified in the parent ProcessComponent. The child ProcessComponent will have a PropertyHolder, itself child of a Generalization whose parent must be the PropertyHolder in the parent ProcessComponent. The child PropertyHolder may add new PropertyDefinition, or PropertyDefinition with the same name of those in the parent PropertyHolder. In the later case, it will be considered an override. When deriving the 'full descriptor' of the child PropertyHolder Class, the specification of the PropertyDefinition in the child will take precedence over the specification of the PropertyDefinition of the parent PropertyHolder.

5.5.10 «enumeration» Granularity

Semantics

Corresponds to the acceptable tagValues for 'granularity' in «ProcessComponent».

Values

Program

Owned

Shared

5.6 Protocol «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.6.1 Virtual metamodel

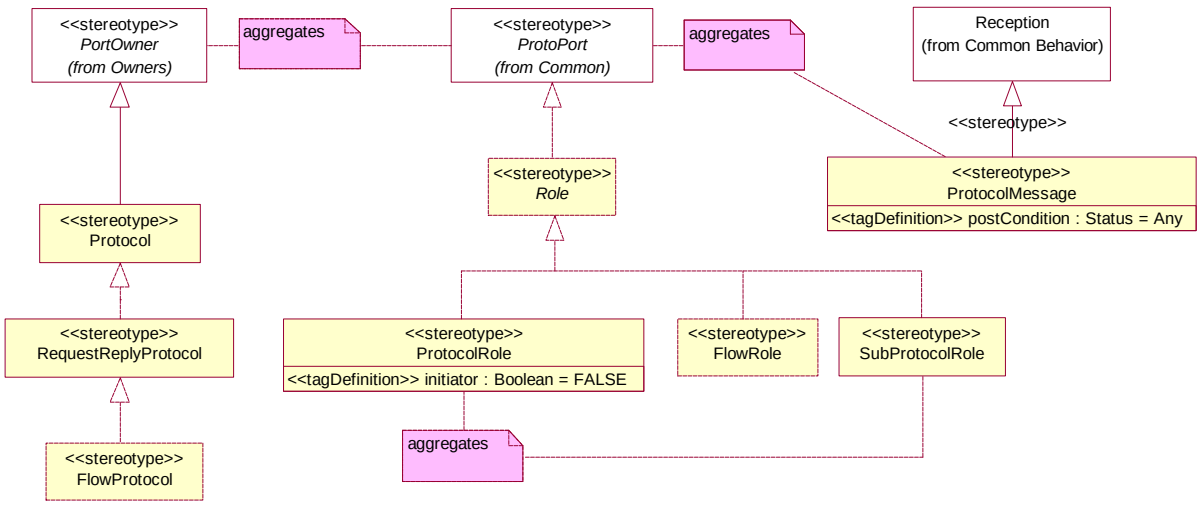


Figure 20: Class Diagram of the Virtual metamodel for Protocol «profile» Package

5.6.2 Applicable subset

From Model Management

?? Subsystem – stereotyped as Protocol, RequestReplyProtocol and FlowProtocol

From Foundation::Core

?? Class – stereotyped as Role, ProtocolRole, FlowRole and SubProtocolRole

From Behavioral Elements::Common Behavior

?? Reception – stereotyped as ProtocolMessage

5.6.3 Accessed Packages

The Protocol «profile» Package accesses the Owners and Common «profile» Packages.

5.6.4 Rationale

A Protocol is a Stereotype of Subsystem, containing Stereotypes of Class specifying the roles of the Protocol.

Role is an abstract supertype to provide common a ancestry for the various role kinds.

ProtocolRole and FlowRole are the roles of the Protocol, and specify the messages that may flow between parties.

ProtocolRole allows any kind of interactions, and may contain a number of ProtocolMessage.

A ProtocolMessage is a Stereotype of the Reception BehavioralFeature, and specifies the capability to receive a Signal with an Attribute typed as a DataElement, and the capability to react to this, by raising one among a set of Signals, each one with an Attribute typed as a different DataElement.

The special RequestReplyProtocol is constrained for simple bi-directional interactions.

The special FlowProtocol is constrained for protocols with a single flow of information.

FlowProtocol and FlowRole does not exist in CCA Conceptual Meta-Model. They are introduced here in support of FlowPort.

FlowRole exists only within FlowProtocol, and constrained to have a single message, while its party (the "other" role in its protocol) will have none.

SubProtocolRole does not exist in CCA Conceptual Meta-Model, and has been introduced to support the concept of SubProtocol of the CCA Conceptual Meta-Model.

SubProtocolRole allows to nest other Protocol as a sub-Protocol, by nesting it into a ProtocolRole. Only Protocol can have SubProtocols, RequestReplyProtocol and FlowProtocol are simpler cases that are not allowed to have SubProtocol.

5.6.5 «Protocol»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«PortOwner»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Inherits from «PortOwner» the capability to own «ProtocolRole» and «FlowRole» .

Standard UML Generalization can be used to produce a more specific Protocol, by specialization of a more generic one. The Protocol child of the Generalization will inherit the various Role of the Generalization parent Protocol.

5.6.6 «Role»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«ProtoPort»	Abstract

Semantics

There is no model element of the same name in the CCA Conceptual Meta-Model.

«Role» has been introduced in the UML Profile for CCA, to provide a common ancestor to «ProtocolRole», «FlowRole» and «SubProtocolRole».

The CCA Conceptual Meta-Model does not need this common ancestor, as it does not specify explicit model elements for «FlowRole» and «SubProtocolRole».

These have been introduced in the UML Profile for CCA in support of the FlowPort and SubProtocol model elements of the CCA Conceptual Meta-Model. Please read their specific headers for details.

When using standard UML Generalization, to produce a more specific Protocol, by specialization of a more generic one, a standard UML Generalization can be used to extend, in the child Protocol, a Role specified in the parent Protocol. The child Role of the Generalization may define additional ProtocolMessage.

5.6.7 «ProtocolRole»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«Role»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Tagged Values

name = "initiator"

tagType = Boolean multiplicity = 1 tagValue= FALSE

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

5.6.8 «ProtocolMessage»

BaseClass	Supertype	Abstract
Behavioral Elements::Common Behavior::Reception	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The type of the «ProtocolMessage» will be expressed by a Signal, with an Attribute typed as a «CompositeData», a DataType, a User defined DataType, or an enumeration.

A «ProtocolMessage» may specify a number of 'raisedSignal'. A raised Signal must have an Attribute typed as a «CompositeData», a DataType, a User defined DataType, or an enumeration.

Through specification of raised Signals, it is possible to express candidate responses to the reception of a «ProtocolMessage».

Specification of more complex sequencing of «ProtocolMessage» may be done with the «Choreography» Stereotype of ActivityGraph. Please refer to section "Choreography «profile» Package" for details.

Tagged Values

name = "postCondition"

tagType = Choreography::Status multiplicity = 1 tagValue= "Any"

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

5.6.9 «SubProtocolRole»

BaseClass	Supertype	Abstract
Foundation::Core::Class	Port	Concrete

Semantics

There is no model element of the same name in the CCA Conceptual Meta-Model.

It has been introduced in the UML Profile for CCA in support of the SubProtocol concept, of the CCA Conceptual Meta-Model, from where it takes its name. Wherever a SubProtocol would be used in the CCA Conceptual Meta-Model, a «SubProtocolRole» must be used, for compliance to UML and the Profile for CCA.

If a «ProtocolRole» must specify a «Protocol» as its (CCA Conceptual M-M) SubProtocol, then a «SubProtocolRole» must be aggregated into the «ProtocolRole».

Note that «ProtocolRole» inherits from the abstract Stereotype «PortNester», and is thus able to contain other specializations of «ProtoPort», in this case a «SubProtocolRole» .

The «SubProtocolRole» will be bound by a Generalization relationship to one of the «ProtocolRole» of the «Protocol» to be aggregated as sub-Protocol. The «SubProtocolRole» must be the child of the Generalization, and the «ProtocolRole» of the sub-Protocol must be the parent.

This pattern is equivalent to the SubProtocol construct of the CCA Conceptual Meta-Model, and captures all the meta-information, and is more precise, as it allows direct binding to one of the «ProtocolRole» of the sub-Protocol. In the CCA Conceptual Meta-Model a convention was , to match the protocol 'initiator' role, with the corresponding sub-Protocol 'initiator'.

Constraints

In compliance to UML visibility and access rules between Packages, a Protocol with SubProtocolRole must have access to the Protocols that become sub-Protocol through SubProtocolRole.

For each Protocol that becomes a sub-Protocol of a top Protocol, through SubProtocolRole, there must be an access Dependency with the top Protocol as client and the sub- Protocol as provider.

(Constraint defined by UML)

5.6.10 «RequestReplyProtocol»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«Protocol»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

5.6.11 «FlowProtocol»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«RequestReplyProtocol»	Concrete

Semantics

There is no model element of the same name in the CCA Conceptual Meta-Model.

It has been introduced in the UML Profile for CCA in support of the «FlowPort», which is constrained to realize «FlowRole».

A «FlowProtocol» has two «FlowRole». One has a single «ProtocolMessage», the other will have no «ProtocolMessage».

Please read section "ComponentSpecification «profile» Package", header «FlowPort» for related details.

5.6.12 «FlowRole»

BaseClass	Supertype	Abstract
Foundation::Core::Class	«Role»	Concrete

Semantics

There is no model element of the same name in the CCA Conceptual Meta-Model.

It has been introduced in the UML Profile for CCA in support of the «FlowPort.

A «FlowRole» has at most a single «ProtocolMessage», and is contained in a «FlowProtocol».

Please read about «FlowProtocol» immediately above, and section "ComponentSpecification «profile» Package", header «FlowPort» for related details.

5.6.13 Collaboration view of a Protocol

A Collaboration (from UML Package Behavioral Elements::Collaborations) may serve as an alternate representation of a «Protocol», using the Collaboration model elements and notation, but without adding any additional specification information.

The Collaboration will have ClassifierRoles with their base referencing the «ProtocolRole» of the «Protocol».

AssociationRoles and AssociationEndRoles in the Collaboration need to reference as their base to Associations and AssociationEnds.

To allow the representation of a «Protocol» as a Collaboration, such Associations may be created within the «Protocol», with connection AssociationEnds referring as their type to the «ProtocolRoles».

The AssociationEnds must have visibility private to the «Protocol».

5.6.14 «Choreography» of a Protocol

A «Choreography» Stereotype of ActivityGraph can be used to specify, for a «Protocol», the valid sequences of messages and activation of sub-Protocols.

It provides a richer mechanism than the one provided by the 'raisedSignal' of «ProtocolMessage». Please refer to section "Choreography «profile» Package" for details.

5.7 ComponentRealization «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.7.1 Virtual metamodel

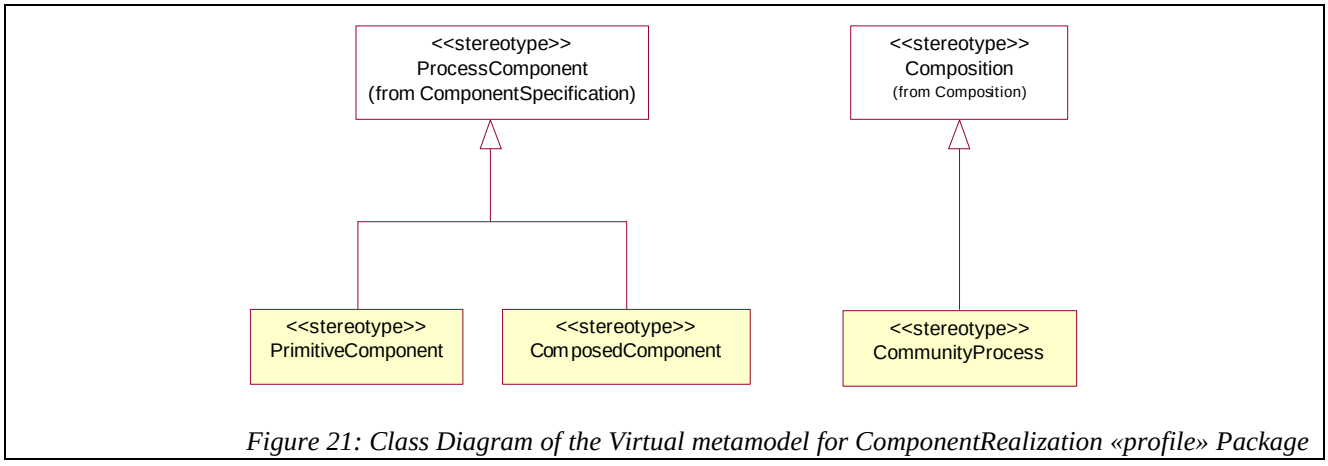


Figure 21: Class Diagram of the Virtual metamodel for ComponentRealization «profile» Package

5.7.2 Applicable subset

From Model Management

?? Subsystem – stereotyped as PrimitiveComponent, ComposedComponent and CommunityProcess

5.7.3 Accessed Packages

The ComponentRealization «profile» Package accesses the ComponentSpecification and Composition «profile» Packages.

5.7.4 Rationale

PrimitiveComponent, ComposedComponent and CommunityProcess are Stereotypes of Subsystem.

PrimitiveComponent is constrained, such that it can not have an internal Composition, but rather refers to a non-CCA artifact as the specification of its realization.

ComposedComponent is the only concrete kind of component, that may actually have an internal Composition. The compositionspecifies the realization of the ComposedComponent in terms of an assembly of other components.

CommunityProcess is just a Composition, constrained such that it does not need, and does not have PortProxies.

5.7.5 «PrimitiveComponent»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«ProcessComponent»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

A PrimitiveComponent will not have internal Composition. Rather it will specify or delegate its actual implementation to non CCA artifacts (i.e. native code, or other UML constructs).

Tagged Values

name = "implementationType"

tagType = String multiplicity = 1 tagValue=

Corresponds to the meta-attribute of the same name in the CCA Conceptual Meta-Model.

name = "implementationLocation"

tagType = String multiplicity = 1 tagValue=

Corresponds to the meta-attribute of the same name in the CCA Conceptual Meta-Model.

5.7.6 «ComposedComponent»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«ProcessComponent»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Note that while in the CCA Conceptual Meta-Model, ComposedComponent directly specializes Composition, in the UML Profile for CCA, «ComposedComponent» is a «CompositionOwner», and contains «Composition».

Because of this, «ComposedComponent» does not directly contain «PortProxy», which are actually contained by its internal «Composition».

5.7.7 «CommunityProcess»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«Composition»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

A «CommunityProcess» inherits from «Composition» the ability to have «ComponentUsage», «Connection» and «PortProxy».

A «CommunityProcess» is constrained such that it must not have «PortProxy». A «PortProxy» is used to bind from within a «Composition», to the external «Port» of its container «ComposedComponent». As a «CommunityProcess» is not contained within a «ComposedComponent», it does not have «PortProxy».

5.8 Composition «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.8.1 Virtual metamodel

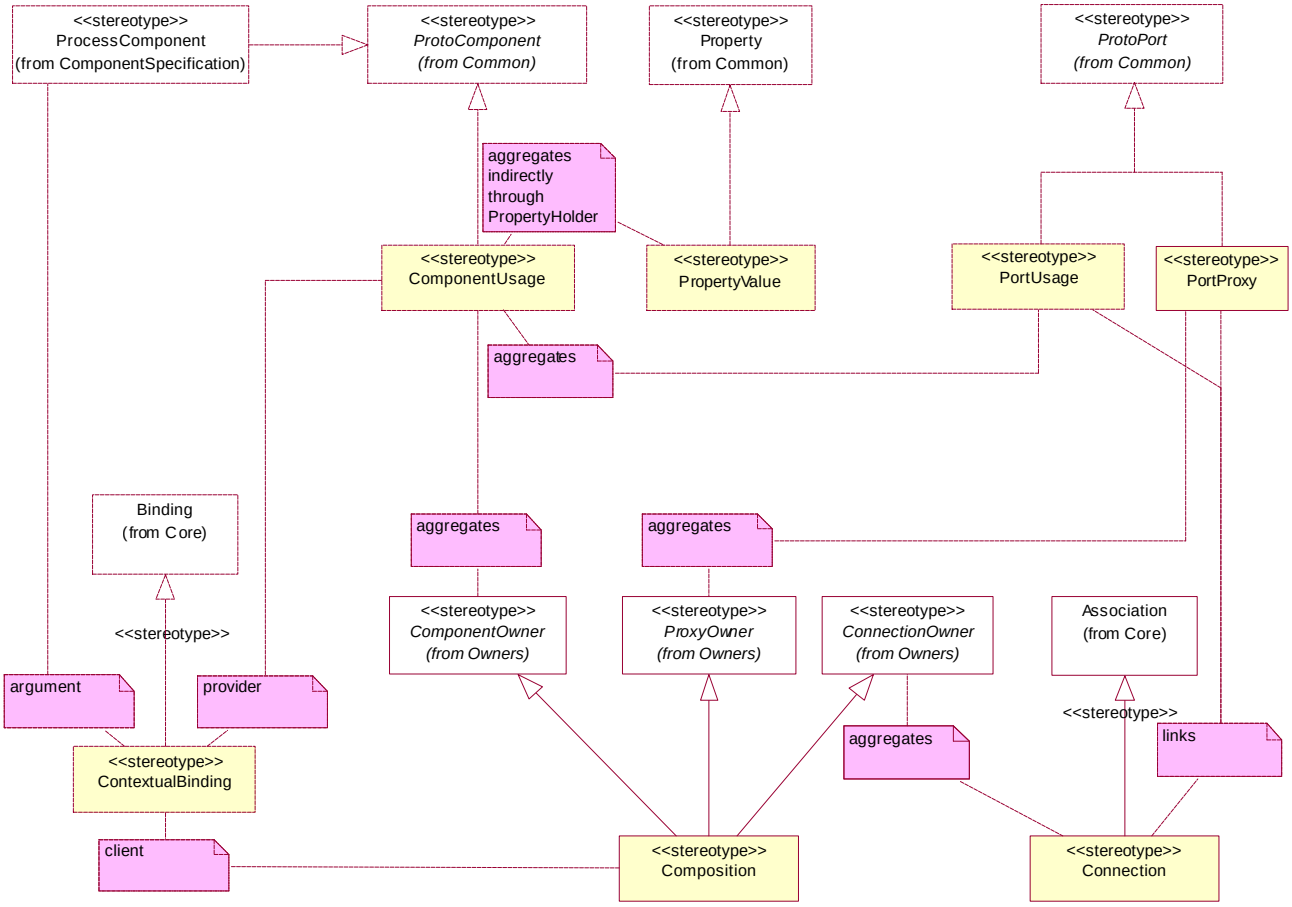


Figure 22: Class Diagram of the Virtual metamodel for Composition «profile» Package

5.8.2 Applicable subset

From Model Management

?? Subsystem – stereotyped as Composition and ComponentUsage

From Foundation::Core

?? Class – stereotyped as PortUsage and PortProxy

?? Attribute – stereotyped as PropertyValue

?? Association - stereotyped as Connection

?? Binding – stereotyped as ContextualBinding

5.8.3 Accessed Packages

The ComponentSpecification «profile» Package accesses the Common and Owners «profile» Packages.

5.8.4 Rationale

A Composition is a Stereotype of Subsystem, where «ProcessComponent» and its «Port» are used as «ComponentUsage» and «PortUsage», respectively. «ComponentUsage» is a Stereotype of Subsystem, and is a specialization of «ProtoComponent», thus having a common ancestry with «ProcessComponent». «PortUsage» is a Stereotype of Class, and is a specialization of «ProtoPort», thus having a common ancestry with «Port».

The «PortUsage» are bound to other «PortUsage» with «Connection», a Stereotype of Association, forming an assembly.

«ProtocolMessage» may flow between the «PortUsage» through the «Connection», according to the «ProtocolRole» realized by the used «Port», and their «Choreography».

If the «Composition» is contained by a «ComposedComponent», then the «Composition» may contain «PortProxy», an Stereotype of Class, specialization of «ProtoPort», thus having a common ancestry with «Port».

«PortProxy» must be bound through «Connection», to the «Port» of the container «ComposedComponent», such that «ProtocolMessage» may flow from and to the «Port» of the container «ComposedComponent», to the «ComponentUsage» of the «Composition».

PropertyValue is a Stereotype of Attribute, used to specify configuration values, in the Composition, for the PropertyDefinition specified on the used ProcessComponent. As the Composition is a Subsystem, and UML constraints prevent a Subsystem from having Attribute, a utility Stereotype of Class, the PropertyHolder, is owned by the ComponentUsage. This is a mechanism identical to the one explained for PropertyDefinition in ProcessComponent.

ContextualBinding is a Stereotype of Binding, to resolve in a Composition, how to substitute the ProcessComponent used by a ComponentUsage, with a different ProcessComponent.

5.8.5 «Composition»

BaseClass	**Abstract**
Model Management::Subsystem	Concrete

Supertypes

?? ComponentOwner – so it can contain Component

?? ConnectionOwner – so it can contain Connection

?? ProxyOwner – so it can contain PortProxy

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

As a difference with the CCA Conceptual Meta-Model, the Composition in the UML Profile for CCA, is not inherited by ComposedComponent, but rather, a ComposedComponent will contain a Composition.

Constraints

In compliance to UML visibility and access rules between Packages, the Composition must have access to the ProcessComponent used by each ComponentUsage in the Composition.

For each ProcessComponent used by ComponentUsage in the Composition, there must be an access Dependency with the Composition as client and the ProcessComponent as provider.

(Constraint defined by UML)

5.8.6 «ComponentUsage»

BaseClass	Supertype	Abstract
Model Management::Subsystem	«ProtoComponent»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

ComponentUsage and ProcessComponent share the common ancestor ProtoComponent.

To specify the reference 'uses' in the CCA Conceptual Meta-Model, from a ComponentUsage, to the ProcessComponent used in the Composition, the UML Profile for CCA utilizes a standard Generalization, with the Generalization parent being the used ProcessComponent, and the Generalization child the ComponentUsage.

A ComponentUsage may own a Stereotype of Class, named PropertyHolder, itself owning PropertyValue, to configure values for the specific conditions and intended behavior of the ProcessComponent in the specific usage in the Composition.

5.8.7 «PropertyValue»

BaseClass	Supertype	Abstract
Foundation::Core::Attribute	«Property»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The attribute 'value' in the CCA Conceptual Meta-Model corresponds in the UML Profile for CCA, to the 'initialValue' metaattribute of Attribute.

To specify in a ComponentUsage, a value with a PropertyValue, for a PropertyDefinition of the same name, in the used ProcessComponent, a PropertyHolder Stereotype of Class must be created and owned by the ComponentUsage.

The PropertyHolder in the ComponentUsage must be the child of a Generalization relationship whose parent will be the PropertyHolder in the used ProcessComponent.

By having the same name in the PropertyDefinition and PropertyValue – both Stereotype of Attribute -, the PropertyValue will be considered an override of the PropertyDefinition.

Both PropertyDefinition and PropertyValue must have the same 'type' and multiplicity.

Only the 'initialValue' metaattribute may differ, and the one in PropertyValue will take precedence when obtaining the 'full descriptor' of the PropertyHolder Class, and therefore will determine the actual value to initialize the property for the ComponentUsage.

5.8.8 «PortUsage»

BaseClass	**Supertype**	**Abstract**
Foundation::Core::Class	«ProtoPort»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

To specify the reference 'represents' in the CCA Conceptual Meta-Model, from a PortUsage in the ComponentUsage, to the Port of the used ProcessComponent, the UML Profile for CCA utilizes a standard Generalization, with the Generalization parent being the Port in the used ProcessComponent, and the Generalization child the PortUsage in the ComponentUsage.

5.8.9 «PortProxy»

BaseClass	**Supertype**	**Abstract**
Foundation::Core::Class	«ProtoPort»	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

To specify the reference 'represents' in the CCA Conceptual Meta-Model, from a PortProxy in a Composition in a ComposedComponent, to an external Port of the enclosing ComposedComponent, the UML Profile for CCA utilizes standard Generalizations, with the Generalization child being the PortProxy, and the Generalization parents being the "conjugate" Roles, of all the Roles realized by the external Port.

"Conjugate" Role is meant as in the Real-Time Object Oriented Method (ROOM), where for a Port realizing a Role in a Protocol, the "conjugate" is the "other" Role of the Protocol, that is not realized by the Port.

With this approach, PortProxy and its represented Port are "connectable", each one realizing one of the parties of a Protocol. The PortProxy represents, within the Composition, the peer Port of other components, that may eventually be connected to the Port of the enclosing ComposedComponent.

This construct allows to connect to the PortProxy, an internal PortUsage, or other PortProxy, as if they were effectively communicating ProtocolMessage with the eventual peers of the enclosing ComposedComponent.

The Port in the ComposedComponent becomes a transparent "pass-through" for the ProtocolMessage traffic incoming and outgoing in/to the externally connectable peers. (In ROOM terms : the Port of the enclosing ComposedComponent is a relay Port).

5.8.10 «Connection»

BaseClass	Supertype	Abstract
Foundation::Core::Association	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Tagged Values

name = "protocolScope"

tagType = Protocol multiplicity = 1 tagValue=

Corresponds to the Association with AssociationEnd of same name, between «Connection» and «Protocol», in the CCA Conceptual Meta-Model.

name = "messageScope"

tagType = ProtocolMessage multiplicity = 1 tagValue=

Corresponds to the Association with AssociationEnd of same name, between «Connection» and «Message», in the CCA Conceptual Meta-Model.

Constraints

In compliance to UML visibility and access rules between elements in different Packages, the PortUsage in different ComponentUsage have no visibility on the PortUsage in other ComponentUsage.

None of the connection AssociationEnd of a Connection will be navigable.

(Constraint defined by UML)

5.8.11 «ContextualBinding»

BaseClass	Supertype	Abstract
Foundation::Core::Binding		Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The 'context' of the ContextualBinding in the CCA Conceptual Meta-Model is represented by the 'client' of the UML Binding, which is the Composition.

The 'fills' of the ConceptualBinding in the CCA Conceptual Meta-Model is represented by the 'provider' of the UML Binding, which is a ComponentUsage.

The 'bindsTo' of the ConceptualBinding in the CCA Conceptual Meta-Model is represented by the 'argument' of the UML Binding, which is a ProcessComponent.

Constraints

Only «Composition» can contain «ContextualBinding».

The 'client' of a «ContextualBinding» is a «Composition».

The 'provider' of a «ContextualBinding» is a (re) used «ProcessComponent» in a «Composition».

The 'argument' of a «ContextualBinding» is a «ProcessComponent».

5.8.12 Collaboration view of a Composition

A Collaboration (from UML Package Behavioral Elements::Collaborations) may serve as an alternate representation of a «Composition», using the Collaboration model elements and notation, but without adding any additional specification information.

The Collaboration will have ClassifierRoles with their base referencing «PortUsages» of «ComponentUsages» in the «Composition».

If the «Composition» is a «ComposedComponent», the Collaboration will have ClassifierRoles with their base referencing the «PortProxies» of the «Composition».

The AssociationRoles and AssociationEndRoles in the Collaboration, will reference as their 'base' the Coonection, and its AssociationEnds, of the «Composition».

5.9 Choreography «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.9.1 Virtual metamodel

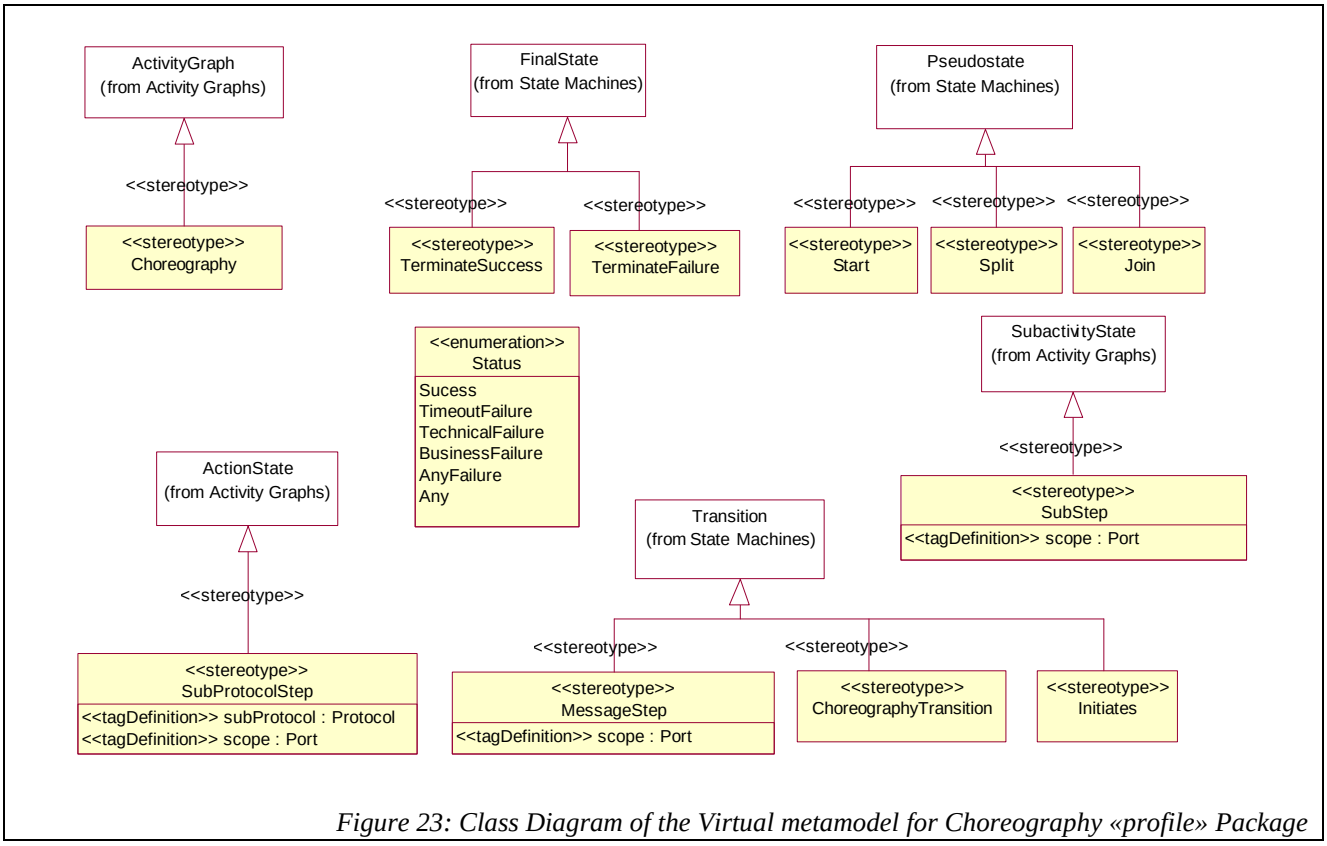


Figure 23: Class Diagram of the Virtual metamodel for Choreography «profile» Package

5.9.2 Applicable subset

From Behavioral Elements::Activity Graphs

?? ActivityGraph – stereotyped as Choreography.

?? ActionState – stereotyped as SubProtocolStep

?? SubActivityState – stereotyped as SubStep

From Behavioral Elements::State Machines

?? Pseudostate – stereotyped as Start, Split and Join

?? FinalState – stereotyped as TerminateSuccess and TerminateFailure

?? Transition – stereotyped as MessageStep and ChoreographyTransition

An enumeration User defined DataType – Status

5.9.3 Accessed Packages

The Choreography «profile» Package accesses the Common «profile» Package.

5.9.4 Rationale

ActivityGraph has been chosen as the baseClass for «Choreography», because it provides the means of specifying the possible sequences of activities and interactions in a system.

FinalState has been chosen as the baseClass for «TerminateSuccess» and «TerminateFailure», as both are special conditions of the termination of an ActivityGraph.

Pseudostate, has been chosen as the baseClass for «Start», «Split» and «Join», with values of its 'kind' metaattribute equal to #initial, #fork and #join, because these are sufficiently similar to the intended semantics.

Transition has been chosen as the baseClass for «MessageStep» because it provides, with an 'effect' or a 'trigger', the means to specify sending or receiving a message.

Transition has been chosen as the baseClass for «ChoreographyTransition» because it provides with a 'guard', the means to specify conditional paths of activity.

ActionState, has been chosen as the baseClass for «SubProtocolStep» because the intention is to express that the interactions of a whole subProtocol will take place as single activity, and an activity is better expressed with an ActionState, and the help of a tagValue to refer to the subProtocol.

SubactivityState, has been chosen as the baseClass for «SubStep», because it allows to nest sub machines, drilling down in each level into more deeply nested scope.

5.9.5 «Choreography»

BaseClass	Supertype	Abstract
Behavioral Elements::Activity Graphs::ActivityGraph	-	Abstract

Semantics

Corresponds to the model element named Choreography in the CCA Conceptual Meta-Model.

Constraints - plain

The 'context' of a «Choreography» is a «Protocol» or a «ProcessComponent», both of them «PortOwner».

A «Choreography» has a Partition (also known as swim-lane) for each «Port» of its 'context' «PortOwner». The name of each Partition will be the name of the «Port» in its 'contents'.

5.9.6 «Start»

BaseClass	Supertype	Abstract
Behavioral Elements::State Machines::Pseudostate		Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Constraints - plain

A «Start» Stereotype of Pseudostate is an Initial state.

5.9.7 «Split»

BaseClass	Supertype	Abstract
Behavioral Elements::State Machines::Pseudostate		Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Constraints - plain

A «Split» Stereotype of Pseudostate is a Fork state.

5.9.8 «Join»

BaseClass	Supertype	Abstract

Behavioral Elements::State Machines::Pseudostate	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Constraints - plain

A «Join» Stereotype of Pseudostate is a Join state.

5.9.9 «TerminateSuccess»

BaseClass	Supertype	Abstract
Behavioral Elements::StateMachines::FinalState	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

5.9.10 «TerminateFailure»

BaseClass	Supertype	Abstract
Behavioral Elements::StateMachines::FinalState	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

5.9.11 «MessageStep»

BaseClass	Supertype	Abstract
Behavioral Elements::State Machines::Transition	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The «MessageStep» may be sent or received.

If the «MessageStep» is in the Partition corresponding to the initiator «Port», then the «ProtocolMessage» is being sent, if not then the «ProtocolMessage» is being received.

Tagged Values

name = "scope"

tagType = SubProtocolRole multiplicity = 0..1 tagValue=

Corresponds to the relationship of the same name in the CCA Conceptual Meta-Model, between Step and StepScope.

The value must be the name of a SubProtocolRole.

If the "scope" taggedValue has been defined, then «ProtocolMessage» whose Signal is referenced as 'effect' SendAction, or 'trigger' SignalEvent, must be one of the «ProtocolMessage» of the SubProtocolRole identified by "scope". ".

Constraints - plain

If the «ProtocolMessage» is being sent, the «MessageStep» will have an 'effect' SendAction, with its 'signal' referencing the Signal of the «ProtocolMessage».

If the «ProtocolMessage» is being received, the «MessageStep» will have a 'trigger' SignalEvent, with its 'signal' referencing the Signal of the «ProtocolMessage».

Diagram Notation

If the «ProtocolMessage» is being sent, a Signal sending symbol for Transition.

If the «ProtocolMessage» is being received, a Signal receipt symbol for Transition.

5.9.12 «SubProtocolStep»

BaseClass	Supertype	Abstract
Behavioral Elements::Activity Graphs::ActionState	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

When producing the Choreography of a «Protocol» with sub-Protocol, there will be a SubProtocolStep for each of the «ProtocolRole» of the Protocol embedded as sub-Protocol.

A Transition stereotyped as «Initiates» must bind with the initiator «SubProtocolRole» as its 'source', and the non-initiator «SubProtocolRole» as its 'target'.

Tagged Values

name = "scope"

tagType = AbstractRole multiplicity = 0..1 tagValue=

Corresponds to the relationship of the same name in the CCA Conceptual Meta-Model, between Step and StepScope.

The value must be the name of the initiator ProtocolRole of the SubProtocol.

name = "subProtocol"

tagType = SubProtocolRole multiplicity = 0..1 tagValue=

Corresponds to the relationship of the same name in the CCA Conceptual Meta-Model, between ProtocolStep and SubProtocol.

The value must be the name of a SubProtocolRole.

If the "scope" taggedValue has been defined, then the "subProtocol" must refer to a subProtocol of the SubProtocolRole identified by "scope"..

5.9.13 «SubStep»

BaseClass	Supertype	Abstract
Behavioral Elements::Activity Graphs::SubactivityState	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

The Step referenced as 'sub' in the CCA Conceptual Meta-Model, will be vertex contained in the submachine ActivityGraph of the SubStep.

If the "scope" taggedValue has been defined, then the Step in the submachine ActivityGraph will resolve names in, and be constrained to, referencing ProtocolMessages and SubProtocolRoles of the SubProtocolRole identified by "scope"..

5.9.14 «ChoreographyTransition»

BaseClass	Supertype	Abstract
Behavioral Elements::StateMachines::Transition	-	Concrete

Semantics

Corresponds to the model element named Transition in the CCA Conceptual Meta-Model.

The guard of the Transition will be an expression that will evaluate true if an specific ProtocolMessage has been actually sent or received.

Tagged Values

name = "precondition"

tagType = Status multiplicity = 1 tagValue=

Corresponds to the attribute of the same name in the CCA Conceptual Meta-Model.

5.9.15 «enumeration» Status

Semantics

Corresponds the Enumeration of same name in the CCA Conceptual Meta-Model.

Values

Success

TimeoutFailure

TechnicalFailure

BusinessFailure

AnyFailure

Any

5.9.16 «Initiates»

BaseClass	Supertype	Abstract
Behavioral Elements::State Machines::Transition	-	Concrete

Corresponds to a Transition between ProtocolStep, in a Protocol with subProtocol in the CCA Conceptual Meta-Model.

Also used as Transition between the ActionStep corresponding to the activities performed on activation of PortUsage or PortProxy, when creating the High Level Activity Graph of a Composition (see section 5.11 "High-level ActivityGraph of a Composition" in page 102).

Semantics

When producing the Choreography of a Protocol with sub-Protocols, will bind to the initiator «SubProtocolRole» as its 'source', and the non-initiator «SubProtocolRole» as its 'target'

When producing an ActivityGraph as alternate representation of a Composition (see section 5.11 "High-level ActivityGraph of a Composition" in page 102), corresponds to a Connection between PortUsage -or PortProxy - in the Composition. The 'source' of the «Initiates» Transition will be the ActionState corresponding to the the activity of the 'initiator' PortUsage, and the 'target' will be the ActionState representing the activity performed by the non-initiator peer connected PortUsage.

5.10 DocumentModel «profile» Package

Corresponds to the package of the same name in the CCA Conceptual Meta-Model.

5.10.1 Applicable subset

The DocumentModel Profile Package identifies the applicable subset of UML elements, within the following accessed UML Packages :

From Foundation::Core

?? Class – stereotyped as CompositeData

5.10.2 Virtual metamodel

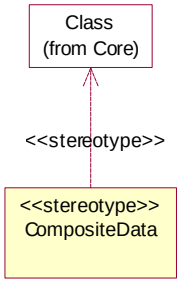


Figure 24: Class Diagram of the Virtual metamodel for DocumentModel «profile» Package

5.10.3 «CompositeData»

BaseClass	Supertype	Abstract
Foundation::Core::Class	-	Concrete

Semantics

Corresponds to the model element of the same name in the CCA Conceptual Meta-Model.

Constraints - plain

The Attributes of a CompositeData will be typed as a DataType, Enumeration or a «CompositeData».

A «CompositeData» may only have supertypes stereotyped as «CompositeData».

A «CompositeData» can not be an active class.

5.11 High-level ActivityGraph of a Composition

An alternate representation of a Composition (i.e., a CommunityProcess or a Component), may be rendered using the model elements of UML ActivityGraph.

Please see example in section 7.1.7 "High level ActivityGraph of a Composition" in page 127.

To produce an ActivityGraph from a Composition, the following constructive rules can be applied :

1. There will be a Partition (also known as swim-lane) for each ProcessComponent in the Composition. The name of the Partition will be the name of the ProcessComponent in the Partition 'contents'.

2. There will be an ActionState (also known as activity) for each Port ,of each ProcessComponent in the Composition, The ActionState will be contents of the Partition associated with the ComponentUsage owning the PortUsage. The name of the ActionState will be the name of the Protocol on the PortUsage (more precisely, the name of the Protocol owning the ProtocolRole realized by the ProtocolPort of the PortUsage).

3. There will be a Transition stereotyped as «Initiates» for each Connection in the Composition, with its 'source' in the ActionState corresponding to an initiator, and its 'target' in the ActionState representing the activity performed by the peer connected PortUsage.

4. If the Protocol of a PortUsage (more precisely: the Protocol owning the ProtocolRole realized by the ProtocolPort used by the PortUsage) has subProtocols, then there will be a SubactivityState, rather than an ActionState, to represent said Port. The SubactivityState will contain a submachine with ActionStates corresponding to each of the subProtocols. Transitions will enter and exit to/from specific sub-activities to represent the various phases of subProtocol activity in the dynamics of the composition.

5. If the Composition pertains to a ComposedComponent, there will be a Partition for each of the PortProxy in the Composition (representing the peers of ProtocolPorts in the enclosing ComposedComponent).

6. If the Composition pertains to a ComposedComponent, there will be an ActionState for each PortProxy, in the corresponding Partition. If the ProtocolRole realized by the PortProxy has sub-ProtocolRole, there will be sub-ActionStates for each of the sub-ProtocolRole. Transitions will enter and exit to/from specific sub-activities to represent the various phases of subProtocol activity in the dynamics of the composition.

5.12 Common «profile» Package

A convenience Package, to assist in the definition of Stereotypes for CCA concepts.

Contain a number of abstract Stereotypes, to be specialized in other Packages.

5.12.1 Virtual metamodel

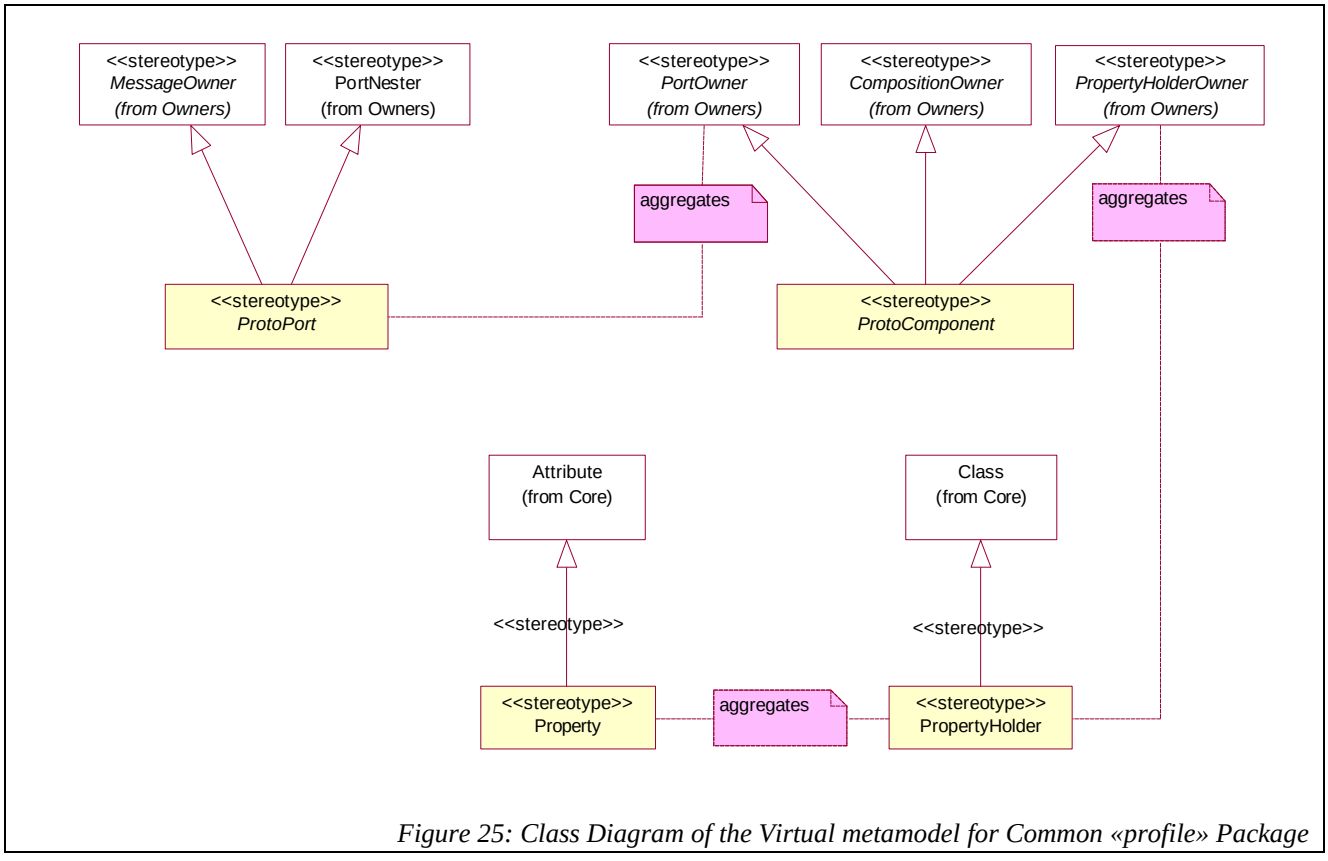


Figure 25: Class Diagram of the Virtual metamodel for Common «profile» Package

5.12.2 Applicable subset

From Model Management

?? Subsystem – stereotyped as ProtoComponent

From Foundation::Core

?? Class – stereotyped as ProtoPort and PropertyHolder

?? Attribute – stereotyped as Property

5.12.3 Accessed Packages

The Common «profile» Package accesses the Owners «profile» Package.

5.12.4 «ProtoPort»

BaseClass		Abstract
Foundation::Core::Class	-	Abstract

Supertypes

MessageOwner – a «ProtoPort» may contain «ProtocolMessage»

PortNester – a «ProtoPort» may contain other «ProtoPort». This capability is used by «ProtocolRole» and «SubProtocolRole», such that it can contain «SubProtocolRole», and thus allowing specification of the concept of SubProtocol in the CCA Conceptual Meta-Model.

Semantics

A common abstract supertype for «ProtocolRole», «Port», «ProtocolPort», «FlowPort», «PortUsage», «PortProxy», all of which may have «ProtocolMessage» – directly or inherited.

To support the SubProtocol meta-model element, in the CCA Conceptual Meta-Model, the facility of «ProtoPort» nesting other «ProtoPort», is introduced here, and exploited by «ProtocolRole» in the Protocol «profile» package.

With the common «ProtoPort» ancestry, Generalizations may be legally specified and constrained, as mapping of the relationships of the CCA Conceptual Meta-Model :

?? 'realizes' «ProtocolRole» with «ProtocolPort» (also «FlowPort», in the profile).

?? 'represents' «ProtocolPort» or «FlowPort» by «PortProxy» or «PortUsage»

Constraints

Only «PortOwner» and its specializations may contain «ProtoPort».

5.12.5 «ProtoComponent»

BaseClass		Abstract
Model Management::Subsystem	-	Abstract

Supertypes

PortOwner – a «ProtoComponent» may contain «Port»

CompositionOwner – a «ProtoComponent» may contain «Composition»

PropertyHolderOwner - – a «ProtoComponent» may contain «PropertyHolder», itself a container for «Property»

Semantics

A common abstract supertype for «ProcessComponent», «ComposedComponent», «PrimitiveComponent» and «ComponentUsage», all of which may have kinds of «ProtoPort» – directly or inherited – and configuration properties/values.

Having the «Composition» containment at this single common supertype simplifies the constraints for Generalizations among more specialized Stereotypes. An OCL constraint in «ProcessComponent» and «PrimitiveComponent» exclude their inherited Composition containment capabilities.

With the common «ProtoComponent» ancestry, Generalizations may be legally specified and constrained, as mapping of the relationships of the CCA Conceptual Meta-Model :

?? 'supertype' between «ProcessComponent» and more specific stereotypes

?? 'uses' «ProcessComponent», «PrimitiveComponent» or «ComposedComponent» by «ComponentUsage»

Constraints

Only «ComponentOwner», its specializations, Model Management::Package and Model Management::Model may contain «ProtoComponent».

5.12.6 «PropertyHolder»

BaseClass	**Supertype**	**Abstract**
Foundation::Core::Class	-	Concrete

Semantics

Serves to hold the Stereotype of Attribute named «Property», within «ProtoComponent», a Stereotype of Subsystem, which UML constrains and can not have Attribute.

More specifically, «ProcessComponent», «ComposedComponent», «PrimitiveComponent» may contain «PropertyHolder» with «PropertyDefinition», while «ComponentUsage» may contain «PropertyHolder» with «PropertyValue».

Constraints

Only «PropertyHolderOwner» and its specializations may contain «PropertyHolder».

5.12.7 «Property»

BaseClass	Supertype	Abstract
Foundation::Core::Attribute	-	Abstract

Semantics

A common supertype for «PropertyDefinition» and «PropertyValue», both representing an structural slot of configuration data.

The 'initialValue' metaattribute of Attribute will be used to specify the attribute 'initial' of PropertyDefinition in the ComponentSpecification package of the CCA Conceptual Meta-Model.

The 'initialValue' metaattribute of Attribute will be used to specify the attribute 'value' of PropertyValue in the Composition package of the CCA Conceptual Meta-Model.

Constraints

Only «PropertyHolder» may contain «Property».

A «Property» has public visibility.

5.13 Owners «profile» Package

A convenience Package, to assist in the definition of Stereotypes for CCA concepts.

Contain a number of abstract Stereotypes, to be specialized by Stereotypes in other Packages of the Profile.

These Stereotypes have their names as "xxxOwner" or "xxxNester", with the "xxx" part specifying the kind of their contained artifacts.

This is intended to help in reading the Profile, as UML Stereotypes do not immediately communicate the elements that may be aggregated by them.

Stereotypes elsewhere in the Profile, specialize these abstract "Owner" Stereotypes. Using multiple inheritance from these "Owner" abstract Stereotypes, the actual combined contents of Stereotypes can be readily expressed.

5.13.1 Applicable subset

From Model Management

?? Subsystem – stereotyped as PortOwner, ComponentOwner, ConnectionOwner, ProxyOwner, PropertyHolderOwner and CompositionOwner

From Foundation::Core

?? Class – stereotyped as MessageOwner, PropertyOwner and PortNester.

5.13.2 Accessed Packages

The Owners «profile» Package accesses no other «profile» Packages.

5.13.3 Rationale

Subsystem has been chosen as the baseClass for PortOwner, ComponentOwner, ConnectionOwner, ProxyOwner, PropertyHolderOwner and CompositionOwner, as it provides both organization and classification capabilities thanks to its supertypes Package and Classifier.

Class is the baseClass for MessageOwner, PropertyOwner, able to contain features.

Class is the baseClass for PortNester, as it provides Port containment capabilities to Port, which is an stereotype of Class.

5.13.4 Virtual metamodel

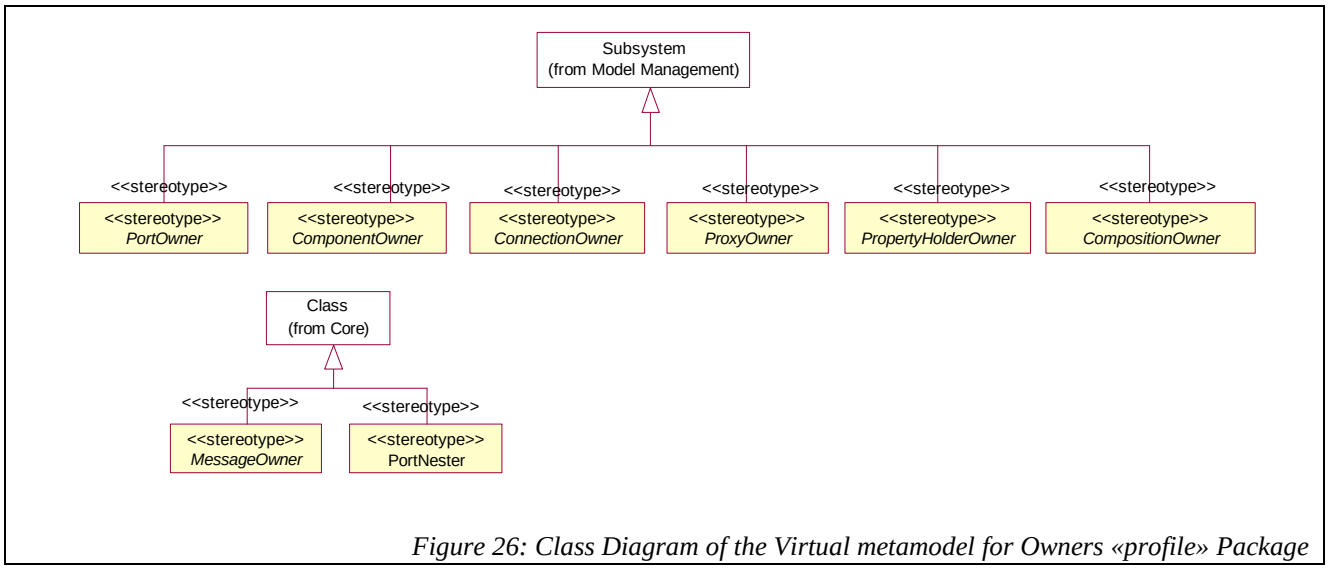


Figure 26: Class Diagram of the Virtual metamodel for Owners «profile» Package

5.13.5 «PortOwner»

BaseClass

Supertype	**Abstract**	
Model Management::Subsystem	-	Abstract

Semantics

Container of «Port»», with Subsystem baseClass

5.13.6 «ComponentOwner»

BaseClass	Supertype	Abstract
Model Management::Subsystem	-	Abstract

Semantics

Container of «ProtoComponent».

5.13.7 «ConnectionOwner»

BaseClass	Supertype	Abstract
Model Management::Subsystem	-	Abstract

Semantics

Container of «Connection».

5.13.8 «ProxyOwner»

BaseClass	Supertype	Abstract
Model Management::Subsystem	-	Abstract

Semantics

Container of «Proxy».

5.13.9 «PropertyHolderOwner»

BaseClass	Supertype	Abstract
Model Management::Subsystem	-	Abstract

Semantics

Container of «PropertyHolder».

5.13.10 «CompositionOwner»

BaseClass	Supertype	Abstract

Model Management::Subsystem	-	Abstract

Semantics

Container of «Composition».

5.13.11 «MessageOwner»

BaseClass	**Supertype**	**Abstract**
Foundation::Core::Class	-	Abstract

Semantics

Container of «ProtocolMessage».

5.13.12 «PortNester»

BaseClass	**Supertype**	**Abstract**
Foundation::Core::Class	-	Abstract

Semantics

Container of «Port», with Class baseClass.

6. Constraints (OCL)

The format for expression of OCL in this document is not (yet) the same as that for the other documents in this submission. This will be corrected in the next revision of this document.

6.1 Invariant Constraints (OCL)

These are the formal OCL constraints specifying well-formedness rules for models according to the UML Profile for CCA.

See section "Definition Constraints", below, for definitions used in these invariants.

6.1.1 ComponentSpecification «profile» Package

6.1.1.1 «Port»

```
context ProtocolPort
    inv:
        not defProtocolRoles->isEmpty()
```

6.1.1.2 «ProtocolPort»

```
context ProtocolPort
    inv:
        defProtocolRoles->forAll( aPR | aPR.isStereoTyped("ProtocolRole"))
```

6.1.1.3 «FlowPort»

```
context FlowPort
    inv:
        defProtocolRoles->forAll( aPR | aPR.isStereoTyped("FlowRole"))
```

6.1.2 Common «profile» Package

6.1.2.1 «ProtoPort»

```
context ProtoPort
    inv:
        not namespace->isEmpty() and namespace.isStereoKinded("PortOwner")
```

6.1.2.2 «ProtoComponent»

```
context ProtoComponent
    inv:
        not namespace->isEmpty() and (
```

```
            namespace.isStereoKinded("ComponentOwner") or
            namespace.isOCLType(Model Management::Package) or
            namespace.isOCLType(Model Management::Model))
```

6.1.2.3 «PropertyHolder»

```
context PropertyHolder
    inv:
        not namespace->isEmpty() and owner.namespace.isStereoKinded("PropertyHolderOwner")
```

6.1.2.4 «Property»

```
context Property
    inv:
        not owner->isEmpty() and owner.isStereoKinded("PropertyHolder") and

    inv:
        visibility = #public
```

6.2 Definition Constraints (OCL)

To improve legibility of constraints in the profile, the following definition constraints are defined in the context of various UML model elements and Profile Stereotypes.

Whenever a token with the name of the definition constraints below is found in a constraint elsewhere in the profile, its value will be derived from the OCL expression in the definition constraint.

6.2.1 General OCL Definition Constraints

These definition constrains have been taken 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 equal to the argument
               -- has been applied to the ModelElement
            self.stereotype.name = theStereotypeName

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

```
            aStereotype.name = theStereotypeName)
```

6.2.2 Protocol «profile» Package

6.2.2.1 «Protocol»

```
context Protocol
   def:
           -- the ProtocolRoles in a Protocol
       let defProtocolRolesStrict : Set( ProtocolRole) =
           ownedElement->select( aModelElement : Foundation::Core::ModelElement |
               aModelElement.isOCLType( Class) and aModelElement.isStereoTyped("ProtocolRole"))

           -- the ProtocolRoles or their specializations, in a Protocol
       let defProtocolRoles : Set( ProtocolRole) =
           ownedElement->select( aModelElement : Foundation::Core::ModelElement |
               aModelElement.isOCLType( Class) and aModelElement.isStereoKinded("ProtocolRole"))
   def:
           -- the set of all immediate parent Protocols
       let defAllImmediateParentProtocols : Set ( Protocol) =
           generalization.parent.oclAsType( ProtocolRole)

   def:
           -- the set of all immediate child Protocols
       let defAllImmediateChildProtocols : Set ( Protocol) =
           specialization.child.oclAsType( Protocol)

   def:
           -- the Association in the Protocol, to support an optional ProtocolCollaboration
       let defAssociation : Association =
           ownedElement->any( aOE : ModelElement | aOE.isOclType( Association))
               .oclAsType( Association)
```

6.2.2.2 «ProtocolRole»

```
context ProtocolRole
   def:
           -- the Protocol of a ProtocolRole
       let defProtocol : Protocol = namespace.oclAsType( Protocol)

   def:
           -- the ProtocolMessages of a ProtocolRole
       let defProtocolMessages : Set( ProtocolMessage) =
           feature->select( aFeature : Foundation::Core::Feature |
               aFeature.isOCLType(Reception) and aFeature.isStereoTyped("ProtocolMessage"))

   def:
           -- all the ProtocolMessages of a ProtocolRole, included inherited ones
       let defAllProtocolMessages : Set( ProtocolMessage) =
           allFeatures->select( aFeature : Foundation::Core::Feature |
```

```
            aFeature.isOCLType( Reception) and aFeature.isStereoTyped("ProtocolMessage"))

    def:
            -- the sole ProtocolMessage of a ProtocolRole
        let defSoleProtocolMessage : ProtocolMessage = defAllProtocolMessages->any( true)

    def:
            -- the conjugate ProtocolRole : the "other" ProtocolRole of its Protocol
        let defConjugateProtocolRole : ProtocolRole =
            defProtocol.defProtocolRoles->any( otherPR : ProtocolRole | not (otherPR = self))


    def:
            -- the Signals of all the ProtocolMessages of the ProtocolRole
        let defAllSignals : Set (Signal) =
            defAllProtocolMessages->collect( aPM : ProtocolMessage | aPM.signal)

    def:
            -- the set of all immediate parent ProtocolRoles
        let defAllImmediateParentProtocolRoles : Set ( ProtocolRole) =
            generalization.parent.oclAsType( ProtocolRole)

    def:
            -- the set of all immediate child ProtocolRoles
        let defAllImmediateChildProtocolRoles : Set ( ProtocolRole) =
            specialization.child.oclAsType( ProtocolRole)
```

6.2.2.3 «ProtocolMessage»

```
context ProtocolRole
    def:
            -- the ProtocolRole of a ProtocolMessage
        let defProtocolRole : ProtocolRole = owner.oclAsType( ProtocolRole)

    def:
            -- the sole/one of the raisedSignal of a ProtocolMessage
        let defSoleRaisedSignal : Signal = raisedSignal->any( true)

    def:
            -- the MessagePayload of the signal of a ProtocolMessage
        let defMessagePayload : MessagePayload =
            signal.feature->any( aF : Feature | aF.isStereoTyped("MessagePayload"))
```

6.2.2.4 «ProtocolPort»

```
context ProtocolPort
    def:
            -- the ProcessComponent of a ProtocolPort
        let defProcessComponent : ProcessComponent = namespace.oclAsType( ProcessComponent)
```

```
            -- the ProtocolRoleRealizations of a ProtocolPort
        let defProtocolRoleRealizations : Set( ProtocolRoleRealization) =
            generalization->collect( aG | Generalization |
                aG.oclAsType( ProtocolRoleRealization))

            -- the ProtocolRoles realized by a ProtocolPort
        let defProtocolRoles : Set( ProtocolRoleRealization) =
            defProtocolRoleRealizations->collect( aPRR | ProtocolRoleRealization |
                aPRR.oclAsType( ProtocolRole))
```

6.2.3 ComponentSpecification «profile» Package

6.2.3.1 «ProcessComponent»

```
context ProcessComponent
    def:
            -- the ProtocolPorts in a ProcessComponent
        let defProtocolPortsStrict : Set( ProtocolPort) =
            ownedElement->select( aModelElement : Foundation::Core::ModelElement |
                aModelElement.isOCLType( Class) and aModelElement.isStereoTyped("ProtocolPort"))

            -- the ProtocolPorts or their specializations, in a ProcessComponent
        let defProtocolPorts : Set( ProtocolRole) =
            ownedElement->select( aModelElement : Foundation::Core::ModelElement |
                aModelElement.isOCLType( Class) and aModelElement.isStereoKinded("ProtocolPort"))

            -- the PropertyHolders in a ProcessComponent
        let defPropertyHolders : Set(PropertyHolder) =
            ownedElement->select( aModelElement : Foundation::Core::ModelElement |
                aModelElement.isOCLType( Class) and
                    aModelElement.isStereotinded("PropertyHolder"))

            -- the PropertyDefinitions in a ProcessComponent
        let defPropertyDefinitions: Set( PropertyDefinition) =
            defPropertyHolders.defPropertyDefinitions
```

6.2.3.2 «Property»

```
context Property
    def:  -- the PropertyHolder of the Property
        let defPropertyHolder : PropertyHolder = owner.oclAsType( PropertyHolder)
```

6.2.3.3 «PropertyHolder»

```
context PropertyHolder
    def:
            -- the ProcessComponent of a PropertyHolder
        let defProcessComponent : ProcessComponent = namespace.oclAsType( ProcessComponent)

            -- the PropertyDefinitions of a PropertyHolder
        let defPropertyDefinitions : Set( PropertyDefinition) =
```

```
            feature->collect( aF : Foundation::Core::Feature |
                aF.oclAsType(PropertyDefinition))
```

6.2.4 Composition «profile» Package

6.2.4.1 «Composition»

```
context Composition
    def:
            -- the ProcessComponents in a Composition
        let defComponentUsage : Set(ProcessComponent) =
            ownedElement->select( aModelElement : Foundation::Core::ModelElement |
                aModelElement.isOCLType( Model Management::Subsystem) and
                    aModelElement.isStereoTyped("ProcessComponent"))

            -- the PortProxy in a Composition
        let defPortProxy : Set( PortProxy) =
            ownedElement->select( aModelElement : Foundation::Core::ModelElement |
                aModelElement.isOCLType(Foundation::Core::Class) and
                    aModelElement.isStereoTyped("PortProxy"))
```

7. Samples

In the sample figures below, various graphical artifacts of the notation, have been annotated with an arrow line, and the name of the virtual metamodel element, or Stereotype, that they represented.

Line and annotation are rendered in blue, when seen in a colour media, or a shade of grey, when media is monochromatic.

These lines and annotations are not part of the proposed notation, but rather just intended help for the understanding of the examples.

7.1 CCA Notation

7.1.1 DocumentModel examples

7.1.1.1 CompositeData definitions

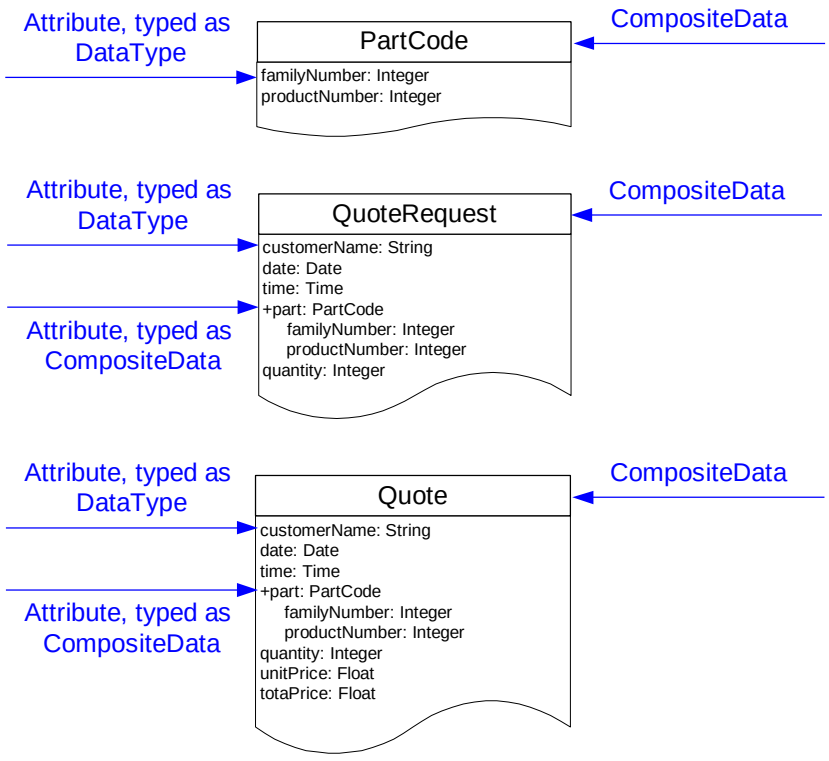


Figure 27 Sample CompositeData definition (CCA)

In this example, a CompositeData named PartCode is defined with two slots of information Attribute, 'familyNumber' and 'productNumber', both type with the Integer DataType.

Another two CompositeData named QuoteRequest and Quote are defined, including Attribute of String, Date, Time and Float DataType, and an Attribute typed as the PartCode CompositeData specified above.

These CompositeData are used in the specification of the ProtocolMessage in the QuoteBT RequestReplyProtocol, below.

7.1.2 Protocol examples

7.1.2.1 Choreographed Protocol

CCA notation for Choreographed Protocols is the same as the UML notation for Activity Graphs – Activity Diagrams, including elements of the notation for State Machines – State Charts. Please, see Protocol examples in the UML Notation section of Examples, below.

A slight difference allowed in CCA is that, to reduce the space used to depict sequences of Sending and Receiving Signals, the sending symbol and the receiving symbols for the response, are positioned one immediately under the other, effectively touching the symbol above. Conversely, the receiving signal symbol, and the send signal symbols for the possible alternative responses, are positioned one under the other, touching.

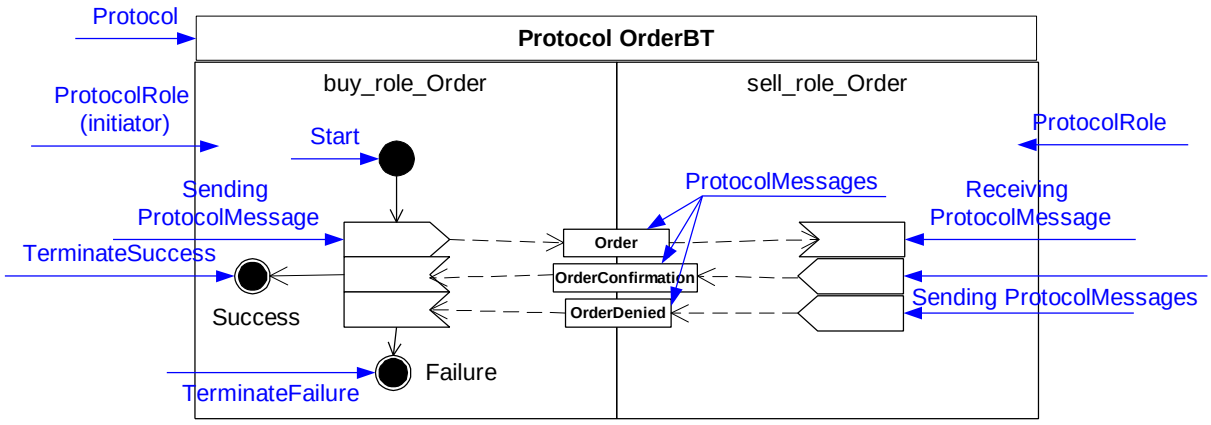


Figure 28: Sample Choreographed Protocol (CCA)

This sample of Protocol using the notation for Activity Diagram, specifies, two ProtocolRoles, buy_role_Order and sell_role_Order, each in its own Partition (swim lane) of the Activity Diagram.

The role buy_role_Order is the initiator, what is shown in its Partition containing the Start initial State.

Roles exchange the ProtocolMessages Order, OrderConfirmation and OrderDenied, with the CompositeData of same name. Sending and receiving of the ProtocolMessage is represented by the UML Symbol for Transitions, Sending and Receiving Signals. A Sending Symbol in one Partition corresponds to the sending of a ProtocolMessage by the

ProtocolRole of the Partition. The Sending (Receiving) Symbol is connected to a Class figure, without compartments, and the name of the CompositeData (or DataType in other examples) carried as attribute of the Signal being sent. The Class figure located in between the Partitions and connected to a Receiving (Sending) Symbol in the opposite Partition.

The sequencing of the ProtocolMessage is shown by the consecutive "touching" layout of the Sending and Receiving Symbols. The two consecutive Sending (Receiving) Symbols represent alternative candidate responses for the ProtocolMessage of the Receiving (Sending) Symbol immediately above. In more standard UML notation for Activity Diagrams, the Sending and Receiving Symbols would be located without touching, and connected by two transition arrows showing two alternate paths of the execution.

Final States, stereotyped as TerminateSuccess or TerminateFailure, are located in the Partition of the initiator ProtocolRole, and represent the alternative candidate outcomes of the Protocol activity.

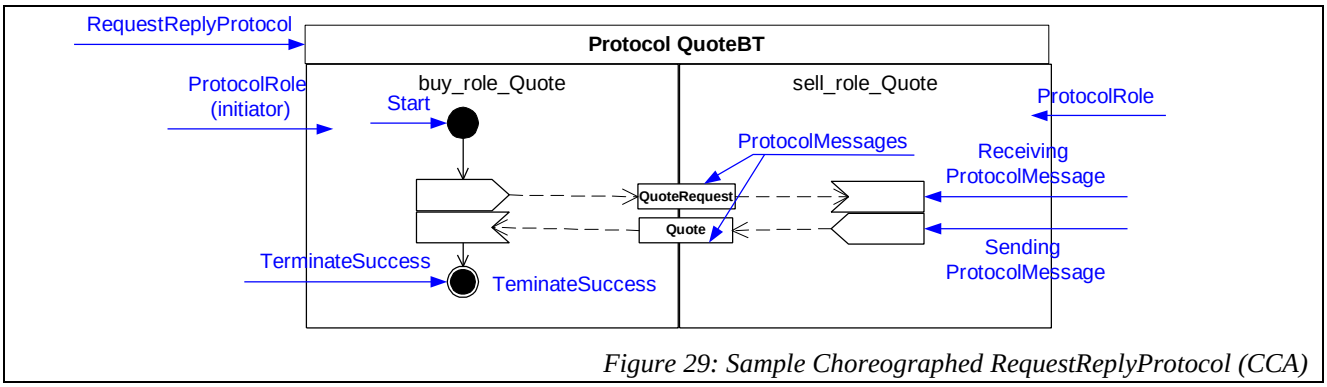


Figure 29: Sample Choreographed RequestReplyProtocol (CCA)

This example of RequestReplyProtocol uses same representation as the Protocol above, with the noticeable difference that there is only one Sending (Receiving) Symbol after the Receiving (Sending) one. Indeed, a RequestReplyProtocol is a case of protocol, constrained specifically to represent this kind of simpler interactions.

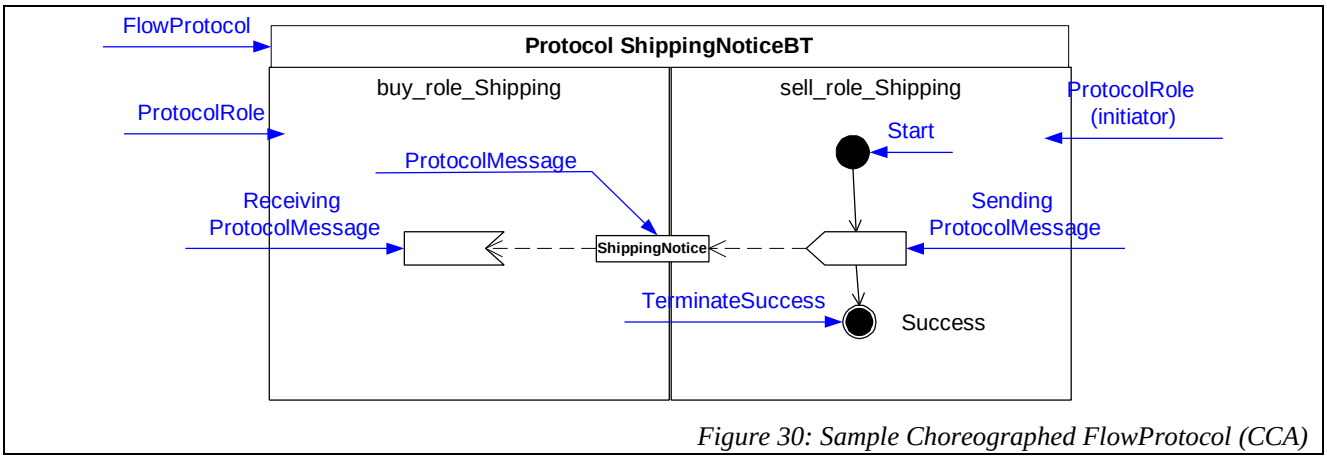


Figure 30: Sample Choreographed FlowProtocol (CCA)

This example of FlowProtocol uses same representation as the Protocol and RequestReplyProtocol above, with the noticeable difference that there is only one Sending and one Receiving Symbol, one in each Partition. According to its constraints, a FlowProtocol is the simpler of the interactions.

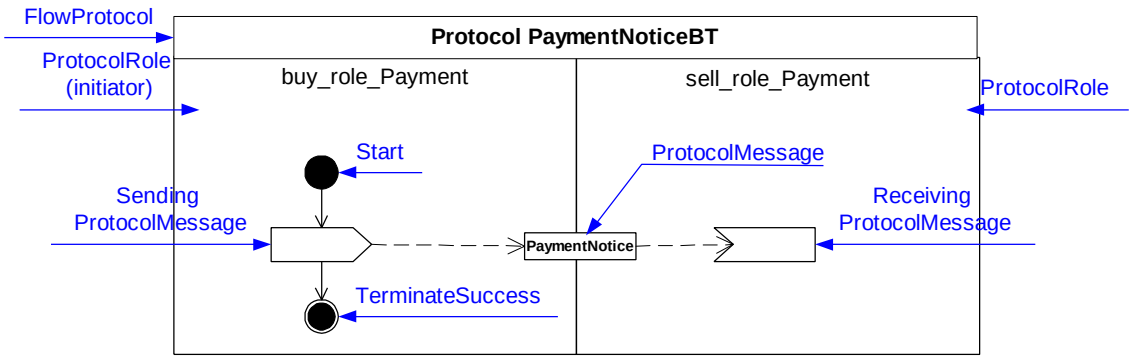


Figure 31: Sample Choreographed FlowProtocol (CCA)

7.1.2.2 Protocol with SubProtocols

CCA notation for Choreographed Protocols is the same as the UML notation for Activity Graphs – Activity Diagrams – , and State Machines – State Charts. SubProtocols are represented by ActionStates with the name of the activated SubProtocol.

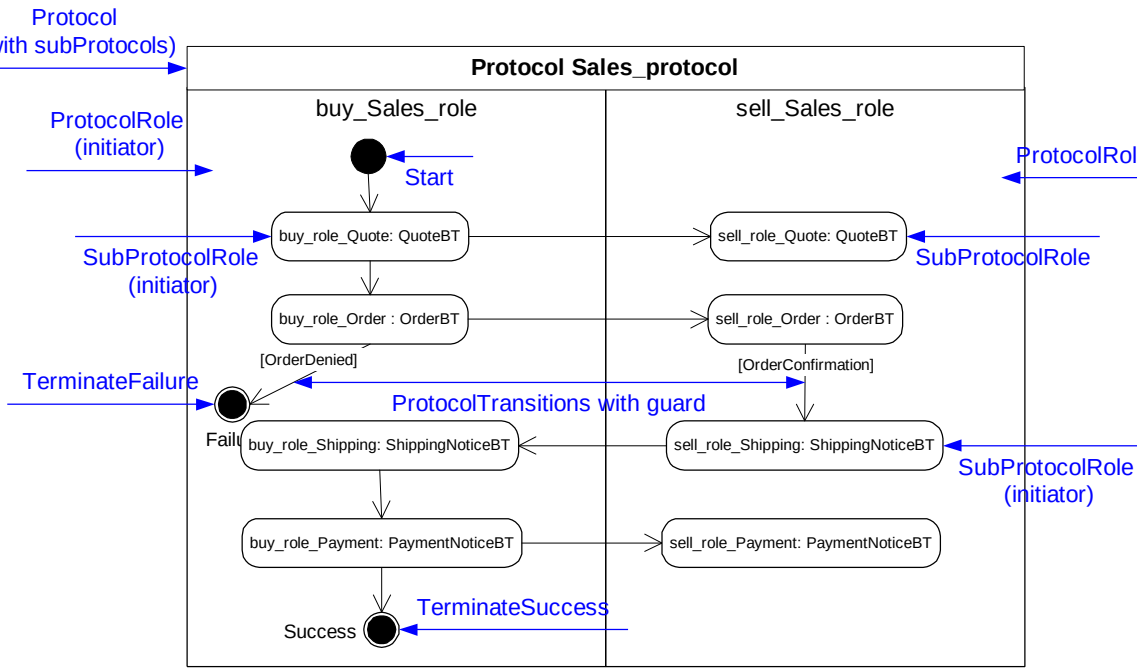


Figure 32: Sample Choreographed Protocol with subProtocols (CCA)

In this example of Protocol with sub-Protocols, the overall activity of the Sales_protocol is specified re-using the more elementary Protocols QuoteBT, OrderBT, ShippingNoticeBT and PaymentNoticeBT.

For each sub-Protocol, two ActionState are inserted, one in Partition of each ProtocolRole, corresponding to each of the ProtocolRole of the sub-Protocol. In the example, the OrderBT Protocol is used as sub-Protocol of the Sales_protocol. For each of the ProtocolRoles in OrderBT, an ActionState is inserted : buy_role_Order in the Partition of buy_Sales_role, and sell_role_Order in the Partition of sell_Sales_role. The represented

semantics are that buy_Sales_role will play the (sub) role of buy_role_Order, when executing activity according to the OrderBT Protocol, conversely, the sell_Sales_role will play the (sub) role of sell_role_Order.

The ActionStates corresponding to a sub-Protocol are connected by a Transition arrow, representing the sequencing dependency between the activity of the initiator ProtocolRole of the sub-Protocol, and the reactive activity of the other ProtocolRole of the sub-Protocol.

Note the Transition arrow from the ActionState for sell_role_Order, in the Partition for the ProtocolRole sell_Sales_role, is connected to the ActionState for sell_role_Shipping, and guarded with the expression [OrderConfirmation]. The represented semantics are that : IF under the activity of sell_role_Order, a ProtocolMessage with an OrderConfirmation flows, THEN the overall activity will proceed with the activity of sell_role_Shipping.

Similarly, a Transition guarded with [OrderDenied] outgoing from buy_role_Order, connects to the TerminateFailure FinalState. The represented semantics is that: IF under the activity of buy_role_Order, a ProtocolMessage with OrderDenied flows, THEN the overall activity will terminate with a failure.

7.1.3 ComponentSpecification examples

7.1.3.1 ProcessComponents

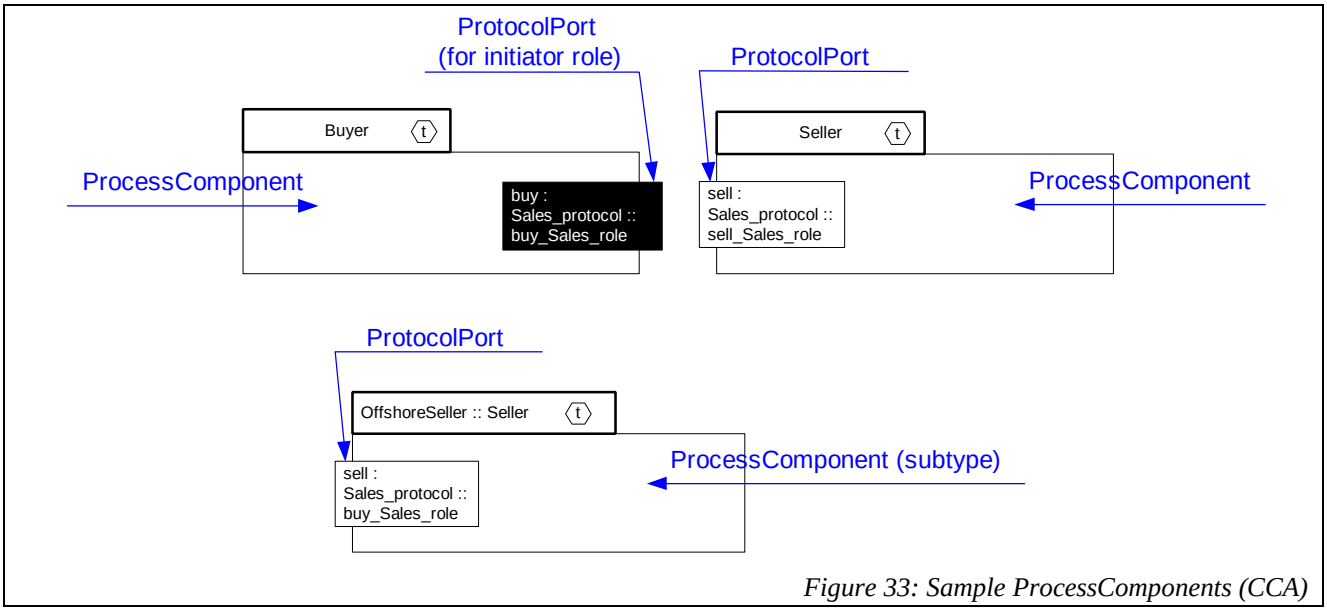


Figure 33: Sample ProcessComponents (CCA)

In this example of ProcessComponent specification, a Buyer ProcessComponent is specified, as having a single buy ProtocolPort, that realizes the initiator ProtocolRole of the Sales_protocol.

The Seller ProcessComponent is specified with the single sell ProtocolPort realizing the non-initiator ProtocolRole of the Sales_protocol.

An specialization of Seller, the OffshoreSeller ProcessComponent is introduced here, and will be referenced in the ContextualBinding example.

Note that no specification is provided about how the ProcessComponent actually perform the activities of the Protocols realized by their ProtocolPorts.

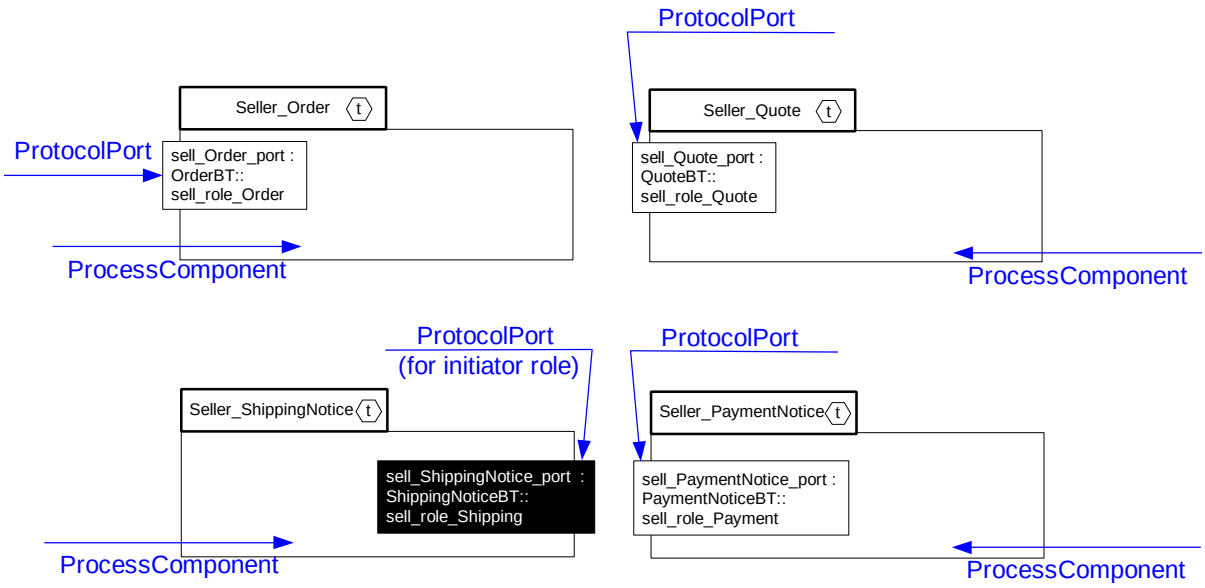


Figure 34: Sample ProcessComponents (CCA) - will be used in the ComposedComponent example

These sample ProcessComponent, Seller_Order, Seller_Quote, Seller_ShippingNotive, Seller_PaymentNotice are specified in a way similar to the examples above. They are defined here for consistency, and will be used in the sample for ComposedComponent.

7.1.4 Composition examples

7.1.4.1 Composition (as a CommunityProcess)

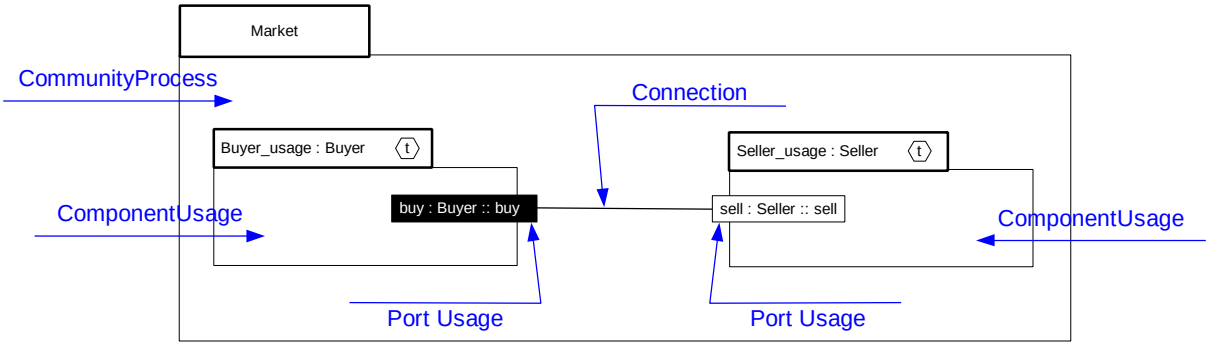


Figure 35: Sample Composition as a CommunityProcess. (CCA)

As an example of Composition, a CommunityProcess is specified, leaving for a later specific example, the case of a Composition in a ComposedComponent.

In the Market CommunityProcess, two ProcessComponent, Buyer and Seller, are used, and incorporated in the Composition as Buyer_usage and Seller_usage, respectively.

The ProtocolPort of the used ProcessComponent are incorporated as PortUsage, in their respective ComponentUsage. Therefore, the Buyer_usage contains a buy PortUsage, corresponding to the buy ProtocolPort of the Buyer ProcessComponent. Similarly, the Seller_usage contains the sell PortUsage corresponding to the sell ProtocolPort of Seller ProcessComponent.

The buy and sell PortUsage are compatible because each is a use of a ProtocolPort realizing complementary ProtocolRole of the same Protocol. Therefore, the ProtocolMessage that can be sent from a PortUsage can be received from the other, and vice versa. Thus it is possible to establish a Connection between the two PortUsage, as rendered in the diagram.

7.1.4.2 ContextualBinding in Community Process

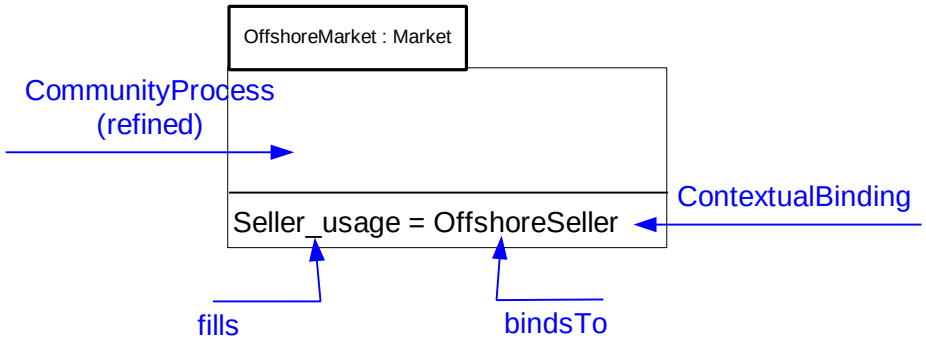


Figure 36: ContextualBinding (in CommunityProcess) (CCA)

In this example for ContextualBinding, a specialization OffshoreMarket, of the Market CommunityProcess above, is specified along with a ContextualBinding.

The OffshoreMarket specifies as its supertype, the previously specified CommunityProcess Market. The refinement introduced by OffshoreMarket is specified by the ContextualBinding.

The ContextualBinding is represented in a separate compartment of the OffshoreMarket CommunityProcess.

Within the OffshoreMarket CommunityProcess, the OffshoreSeller ProcessComponent will fill the Seller_usage, in the Market CommunityProcess, and will be used rather than the one originally used in the Market CommunityProcess.

7.1.5 ComponentRealization examples

7.1.5.1 ComposedComponent

Please be advised that in this example, the CCA notation has been abused, to provide to the reader, directly in the diagram, information that will allow to better trace the elements in the diagram, to elements in other related example diagrams, and the elements in the Conceptual Meta-Model. The notational abuses are :

?? The name of the ComposedComponent is followed by the name of its supertype Component, as in :

"Seller_composed : Seller"

?? The name of the ProtocolPort is followed by the name of the ProtocolRole that it realizes, fully qualified with the name of the Protocol (usually the name of the ProtocolPort is rendered alone), as in :

"sell : Sales_protocol::sell_Sales_role"

?? The name of each ComponentUsage is followed by the name of the Component that is being used (usually the name of the ComponentUsage is rendered alone), as in :

"Seller_Quote_usage : Seller_Quote"

"Seller_Order_usage : Seller_Order"

"Seller_ShippingNotice_usage : Seller_ShippingNotice "

"Seller_PaymentNotice_usage : Seller_PaymentNotice "

?? The name of each PortUsage is followed by the name of the Port that is being used, fully qualified with the name of the Component (usually the name of the PortUsage is rendered alone), as in :

"sell : Seller_Quote :: sell_Quote_port"

"sell : Seller_Order :: sell_Order_port"

"sell : Seller_ShippingNotice :: sell_ShippingNotice_port"

"sell : Seller_PaymentNotice :: sell_PaymentNotice_port"

?? The PortProxy is shown as a distinct, separate box, has been named, and is followed by the name of the ProtocolRole that it realizes, fully qualified with the name of the Protocol (usually the PortProxy is rendered as a small box contiguous to that of the represented ProtocolPort, and the name of the PortProxy is left anonymous and not rendered), as in :

"buy : SalesProtocol :: buy_Sales_role"

?? The elements that pertain to the internal Composition of the ComposedComponent, has been framed in a box with dotter line border (usually the boundary of the Composition is not rendered).

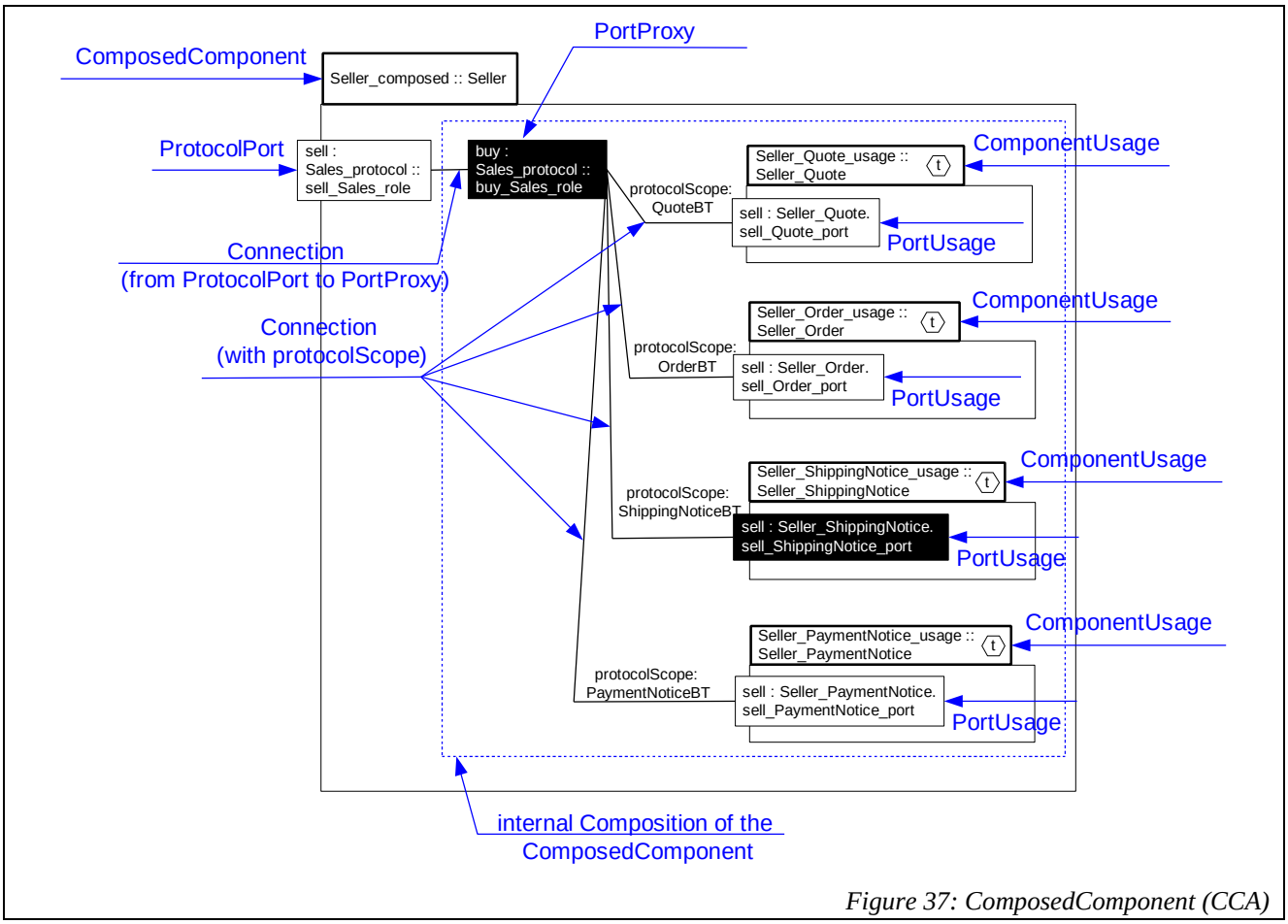


Figure 37: ComposedComponent (CCA)

In this example, the Seller_composed is a ComposedComponent, specified as a subtype of the Seller ProcessComponent previously defined in an example above. Therefore, the Seller_composed is substitutable with Seller, and actually provides a specification of how will be carried out the activities corresponding to the Protocol realized by the ProtocolPort.

Seller_composed has an internal Composition, although it is not separately depicted in the notation, other than by having the model elements of the Composition located inside the box figure of the ComposedComponent.

The Seller_composed ComposedComponent has inside (its Composition), a number of ComponentUsage : Seller_Quote_usage, Seller_Order_usage, Seller_ShippingNotice_usage, Seller_PaymentNotice_usage, each corresponding to uses of the predefined ProcessComponent : Seller_Quote, Seller_Order, Seller_ShippingNotice, Seller_PaymentNotice. Each ComponentUsage has PortUsage corresponding to the ProtocolPort of their used ProcessComponent.

The sell ProtocolPort of the Seller_composed ComposedComponent provides a pass-through (or Relay port, un UML-RT terms), such that the internal ComponentUsage can communicate with the outside of the Seller_composed.

The Composition of the Seller_composed has a PortProxy that represents, within the Composition, the ProtocolPort that may eventually be externally connected to the sell ProtocolPort of the Seller_composed. The PortProxy is named 'buy', and in fact realizes the

ProtocolRole buy_Sales_role, conjugated in the Sales_protocol, to the ProtocolRole sell_Sales_role, realized by the sell ProtocolPort of Seller_composed.

As the sell ProtocolPort, and the buy PortProxy, realize each one of the ProtocolRole of the same Protocol Sales_protocol, it is possible to establish a Connection between the sell ProtocolPort and the buy PortProxy. This is depicted in the example diagram, with the usual UML line for associations.

In this example, the designer has chosen a pattern, where the activities for each of the sub-Protocol, realized by the sell ProtocolPort, is delegated to one internal ComposedUsage. To accomplish this delegation, the buy PortProxy is linked through a Connection, to the sell PortUsage of each of the ComponentUsage.

To ensure that only the ProtocolMessage, corresponding to the sub-Protocol supported by the each of the PortUsage, is delegated through the Connection, each Connection is fine-tuned with the protocolScope taggedValue. The taggedValue is set with the name of the sub-Protocol whose ProtocolMessage are allowed to flow through the Connection.

7.1.6 Choreography examples

7.1.6.1 Choreography of a Protocol

Choreography of Protocols have already been shown in the examples for Protocol, above. Protocols have been shown in their Choreographed notation, as Activity Diagrams.

See Figure 31: Sample Choreographed FlowProtocol (CCA) in page 119, and Figure 32: Sample Choreographed Protocol with subProtocols (CCA) in page 119.

7.1.6.2 Choreography of a ProcessComponent

The previous examples of ProcessComponent have a single ProtocolPort, and the example of Choreography would not be quite illustrative. As the subject for the example of Choreography in a ProcessComponent, a new ProcessComponent is specified, with a number of ProtocolPort.

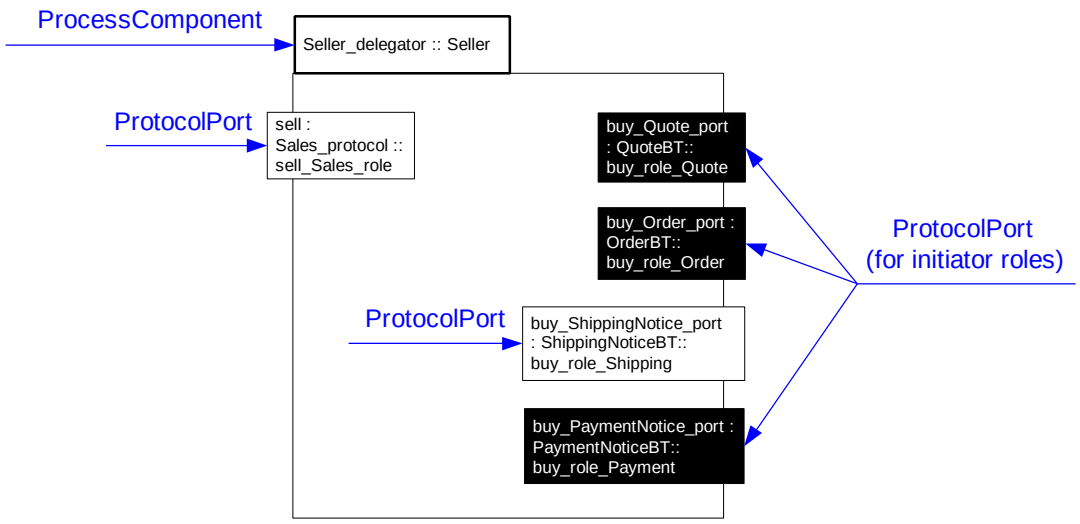


Figure 38: ProcessComponent for example on Choreography of ProcessComponent (CCA)

In this ProcessComponent, the designer has chosen a pattern, where a ProtocolPort realizes the sell_Sales_role of the Sales_protocol, and a number of ProtocolPort realize the buyer-side ProtocolRole, of each of the sub-Protocol of the Sales_protocol.

The ProcessComponent will delegate into external ProcessComponent, the activities corresponding to each of the sub-Protocol.

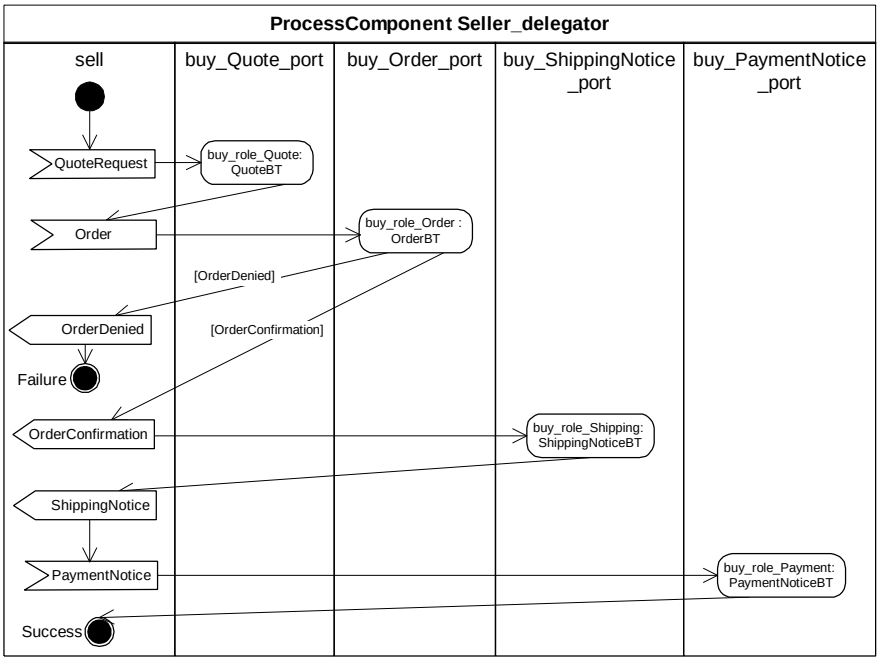


Figure 39: Choreography of ProcessComponent – with sub-Protocols (CCA)

The Choreography of the Seller_delegator ProcessComponent, expressed as an ActivityGraph, specifies the order in which ProtocolMessages will be received and sent,

and when will be activated the Protocol, on ProtocolPort named as "buy_Xxx_port" (with "Xxx" being "Quote", "Order", "ShippingNotice" and "PaymentNotice)..

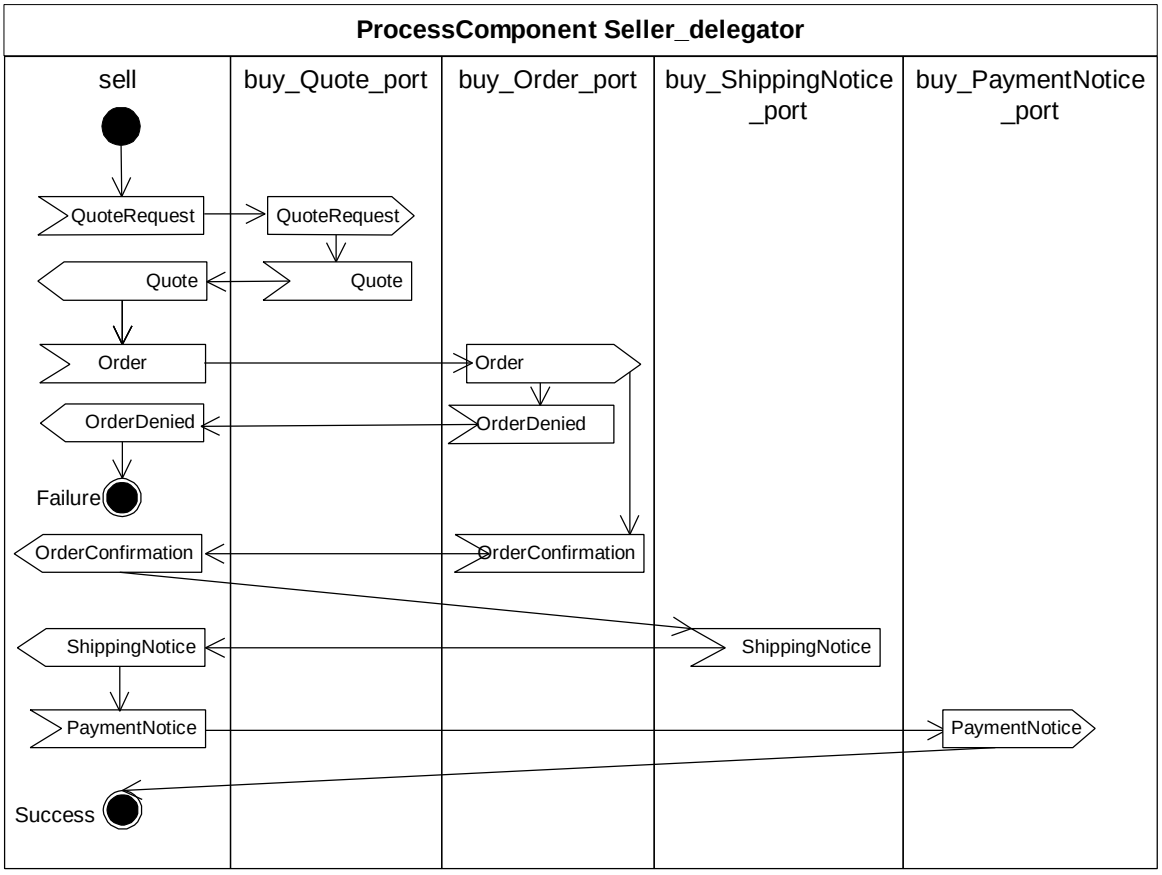


Figure 40: Choreography of ProcessComponent (CCA)

In this expanded ActivityGraph rendering of the Choreography of the Seller_delegator ProcessComponent, the Protocols on the buy_xxx ProtocolPort have been exploded n their individual ProtocolMessage, allowing a more direct perception of the sequences of ProtocolMessages that will be received and sent through each ProtocolPort.

7.1.7 High level ActivityGraph of a Composition

A UML ActivityGraph can be used to provide an alternate representation of the Composition.

The Composition subject of this sample High Level Activity Graph, is the internal Composition of the ComposedComponent described in Figure 37: ComposedComponent (CCA) in page 124.

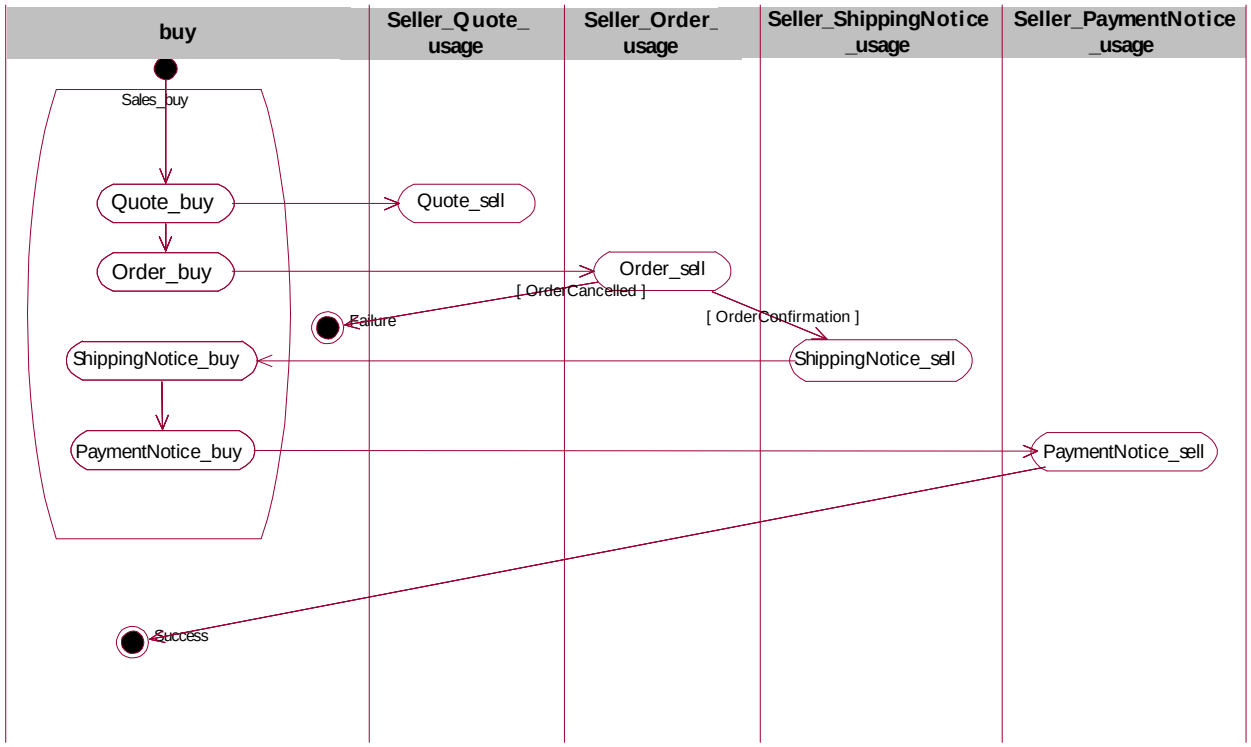


Figure 41: High Level ActivityGraph of Composition (CCA)

In this expanded ActivityGraph rendering of the Choreography of the Seller_delegator, a Partition (swim-lane) has been created for

7.2 UML Notation

7.2.1 DocumentModel examples

7.2.1.1 CompositeData definitions

Standard UML Class diagrams are used to represent the structure of ProtocolMessage information payload. Attributes are rendered in their usual compartment. Note that slots of CompositeData within a container CompositeData Class, is done as Attributes, and not through Associations.

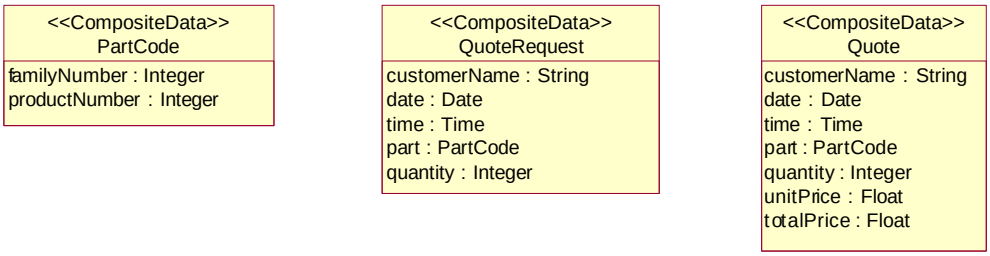


Figure 42: Sample CompositeData definition (UML)

7.2.2 Protocol examples

7.2.2.1 Protocol,RequestReplyProtocol, FlowProtocol

Protocol, ProtocolRole and ProtocolMessage can be rendered in UML both as Class diagrams – a purely structural representation – and as ActivityGraphs, where their Choreography is also represented.

The UML representation of Protocol, ProtocolRole and ProtocolMessage, as Class diagram, is done according to the standard UML notation of the baseClass of their defined Stereotypes.

Note that some tools do not fully support a notation for Reception BehavioralFeature. To overcome this limitation, the compartment and specification for Operation BehavioralFeature is used instead. The ProtocolMessage becomes an Operation, with the name of the Operation corresponding to the name of the ProtocolMessage. An argument of the Operation (in this case with the chose name of 'payload'), serves to capture a reference to the CompositeData attribute of the Reception's Signal.

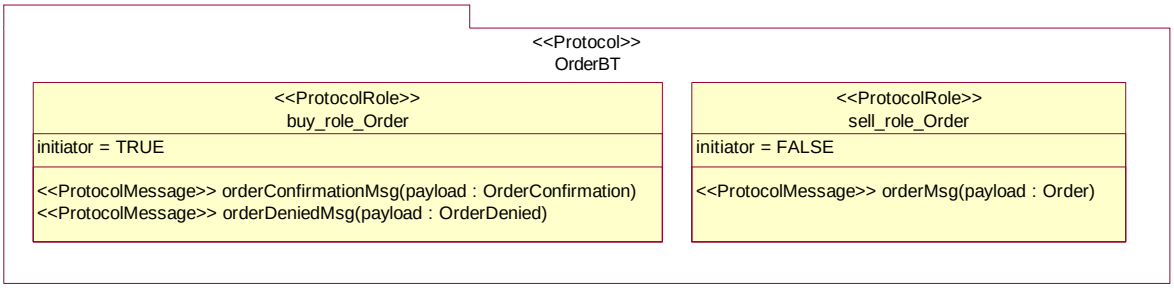


Figure 43: Sample Protocol (UML)

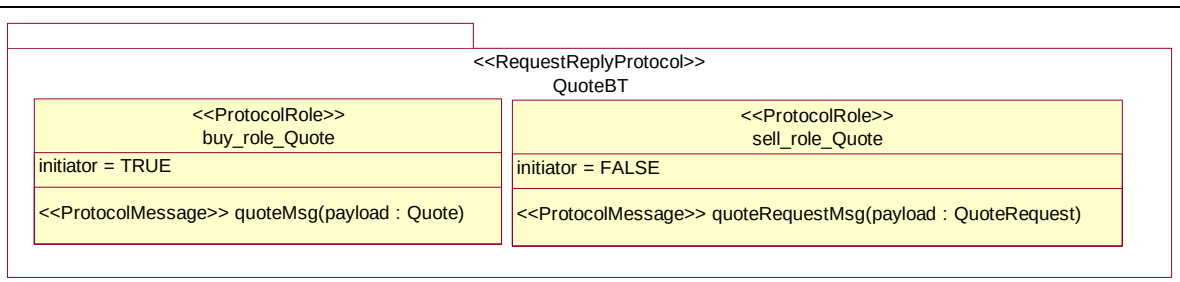


Figure 44: SampleRequestReplyProtocol (UML)

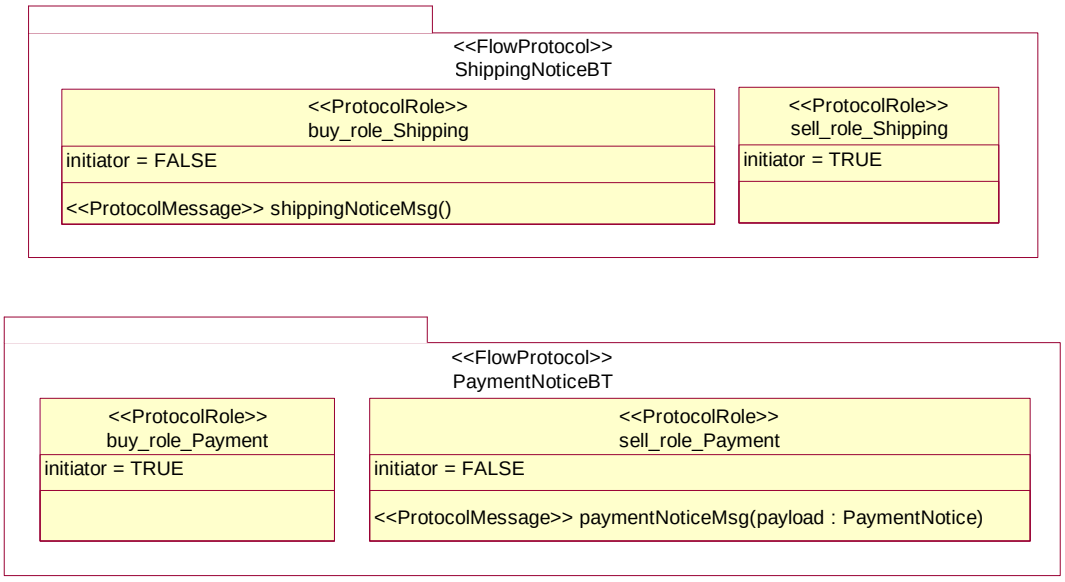


Figure 45: Sample FlowProtocol (UML)

The UML rendering of a Choreographed Protocol is an ActivityGraph. The CCA representation is very similar, with just a small space saving variation. Please see Figure 28: Sample Choreographed Protocol (CCA) in page 117, and immediately following RequestReplyProtocol and FlowProtocol examples.

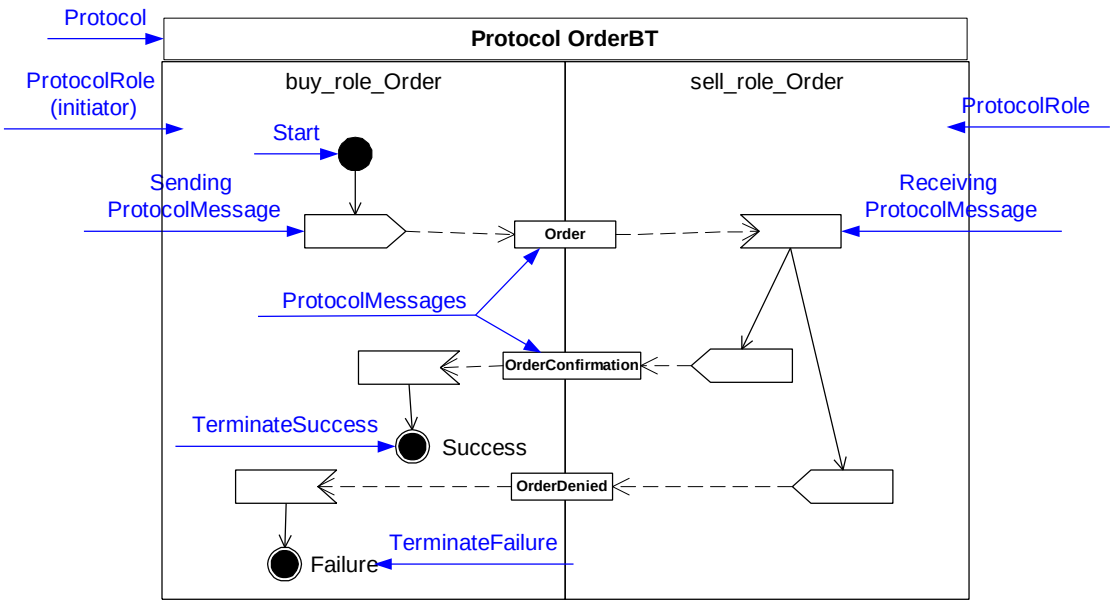


Figure 46: Sample Choreographed Protocol (UML)

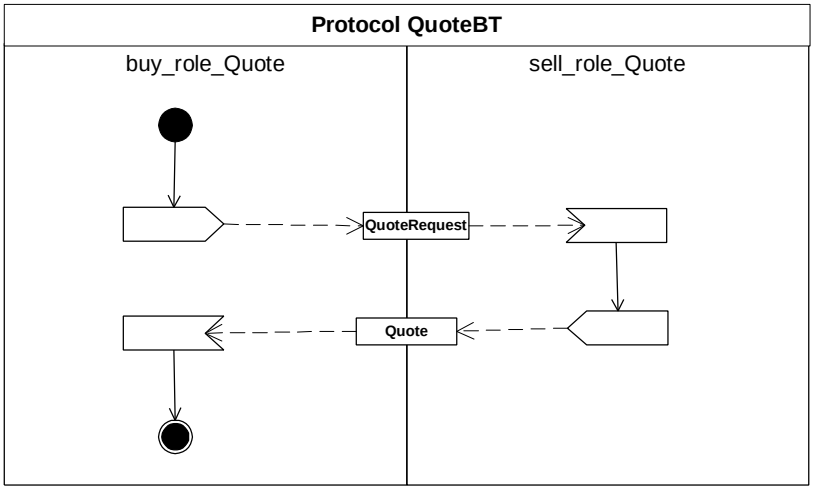


Figure 47: Sample Choreographed RequestReplyProtocol (UML)

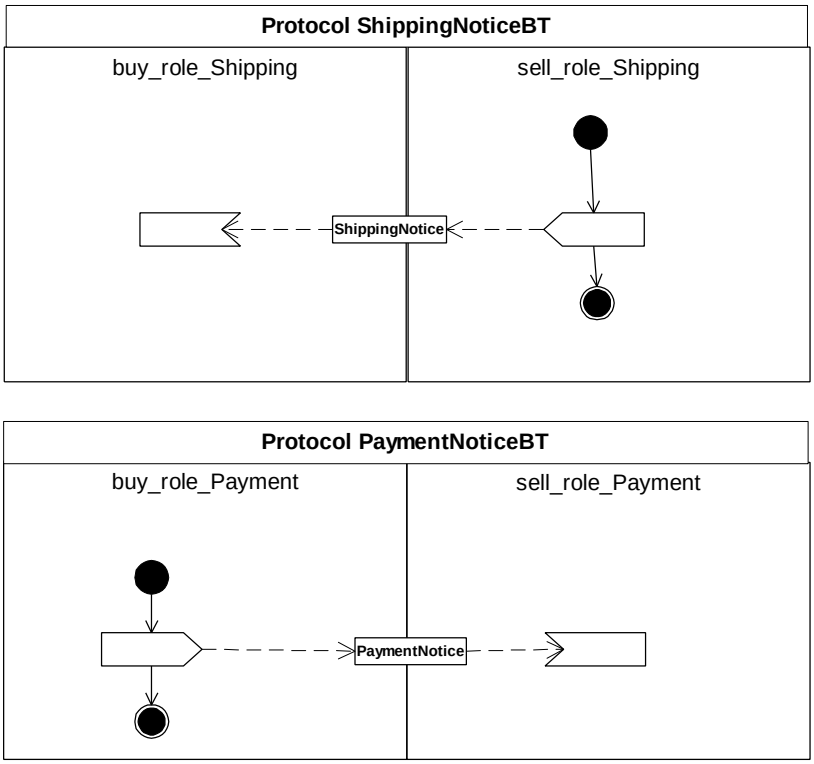


Figure 48: Sample Choreographed FlowProtocol (UML)

Figure 49: Sample Protocol, RequestReplyProtocol, FlowProtocol (UML Collaboration view)

See above a sample rendering of Protocol as Collaboration diagrams (classifier level). The Protocol must be already specified in its structural, and optionally the Choreography ActivityGraph form.

Note that no additional information is added to the specification of the Protocol, and that some partial ordering of ProtocolMessage, that can be expressed by the Choreography ActivityGraph, can not be expressed completely by the mechanisms available in UML Collaborations.

7.2.2.2 Protocol with SubProtocols

See below a rendering of the Protocol Sales_protocol, with sub-Protocol. Aggregation notation is used to represent the nesting of SubProtocolRole, within the ProtocolRole of the top-level Protocol.

The inheritance of the SubProtocolRole, from the ProtocolRole of the sub-Protocol, is made explicit in the diagram below.

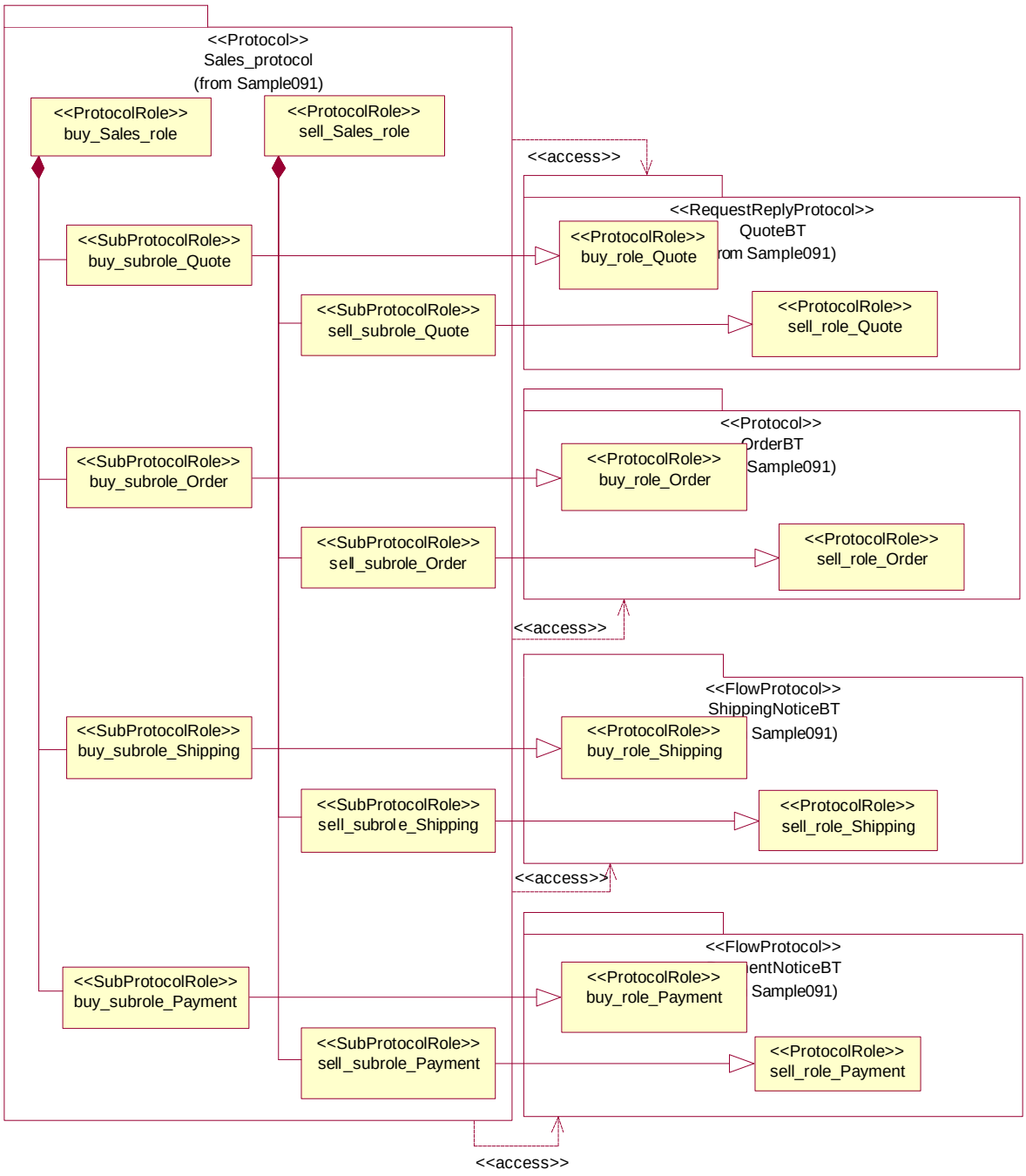


Figure 50: Sample Protocol with SubProtocols (UML)

The UML rendering of a Choreographed Protocol with sub-Protocol is an ActivityGraph, with ActionState representing the activation of sub-Protocol. Its representation in CCA is identical to the standard UML ActivityGraph. Please see Figure 32: Sample Choreographed Protocol with subProtocols (CCA) in page 119.

A Protocol with sub-Protocol may be rendered in an ActivityGraph representation where the flow of ProtocolMessage in the sub-Protocol are exploded and made explicit in the top level Protocol. While this is not an encouraged practice, it may be sometimes useful, for a more immediate perception of the overall ProtocolMessage involved. Following is a sample of such an exploded view of sub-Protocol.

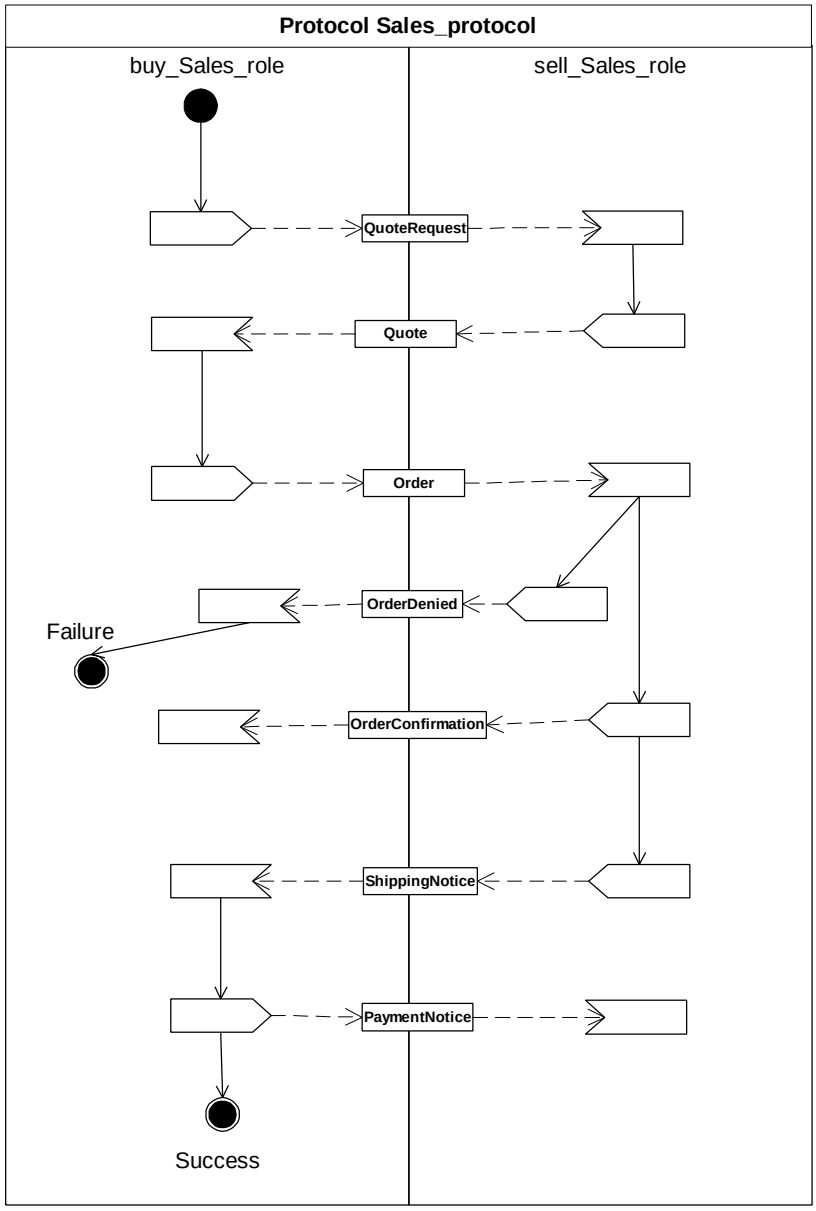


Figure 51: Sample Choreographed Protocol with exploded SubProtocols (CCA)

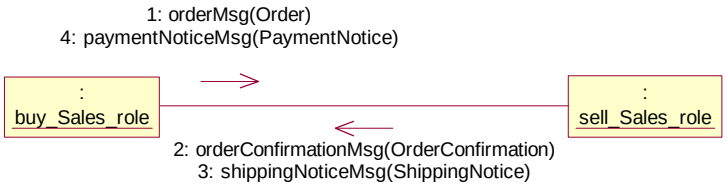


Figure 52: Sample Protocol with SubProtocols (UML Collaboration view)

7.2.3 ComponentSpecification examples

7.2.3.1 ProcessComponents

ProcessComponent specifications can be rendered using conventional UML Class diagrams. Stereotyped Port Classes are shown within the frame of their container Stereotyped ProcessComponent Subsystem.

In the sample diagram below, the ProtocolRole realized by the Port is shown, and the realization relationship made explicit with the standard Generalization notation.

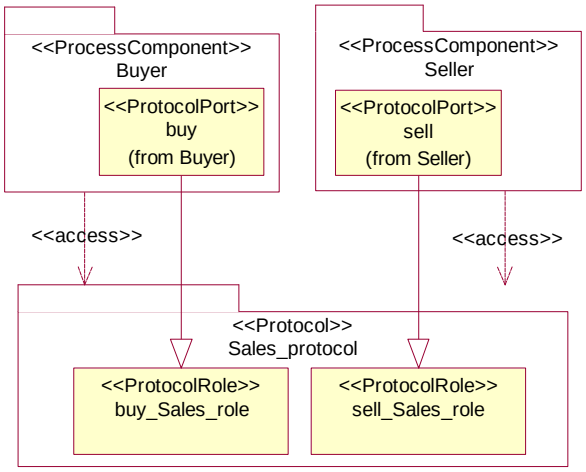


Figure 53: Sample ProcessComponents, with PropertyDefinitions, and ProtocolPorts (UML)

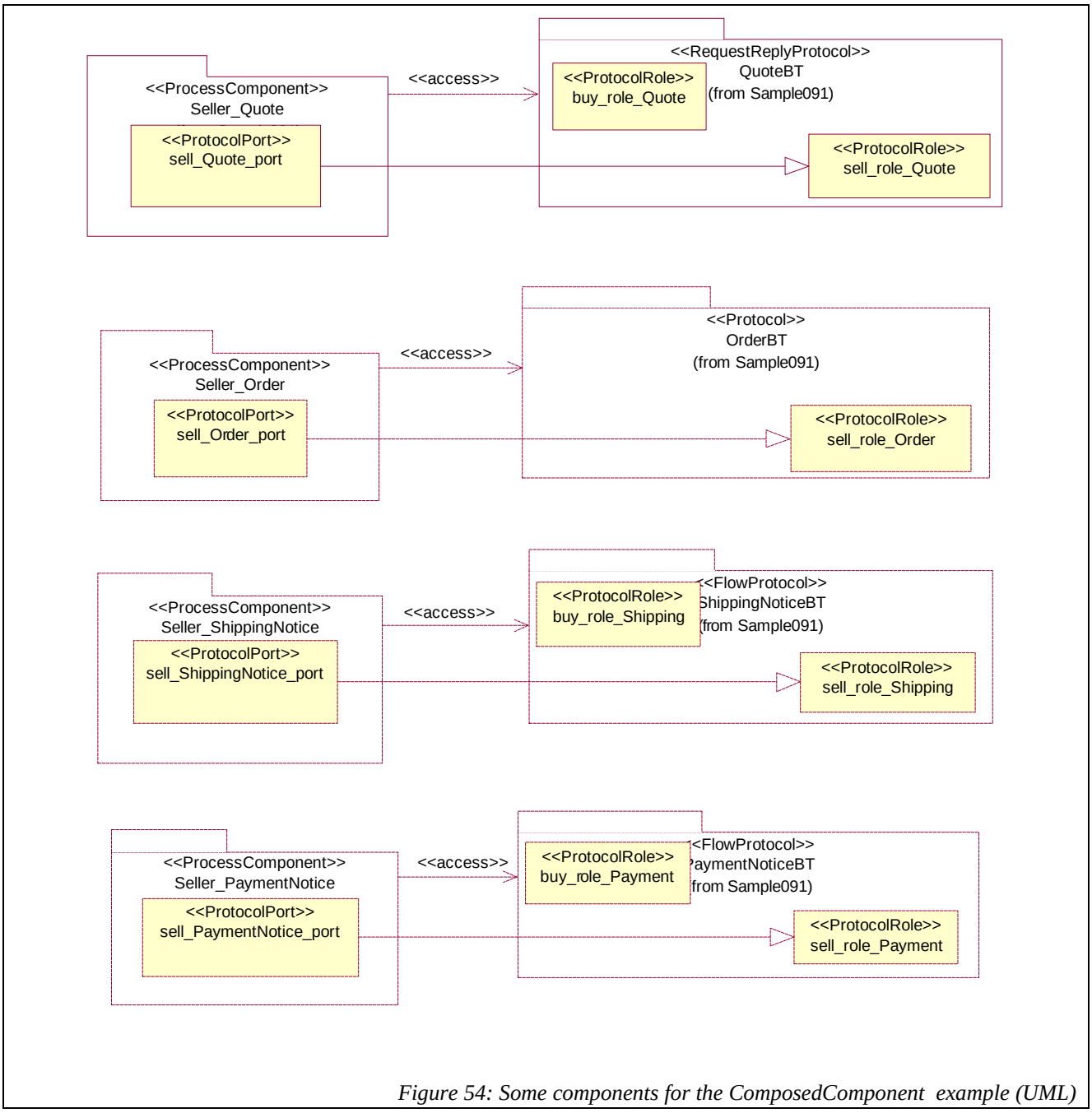


Figure 54: Some components for the ComposedComponent example (UML)

7.2.4 Composition examples

7.2.4.1 Composition (as a CommunityProcess)

Composition specifications, as the CommunityProcess below, can be rendered using conventional UML Class diagrams. Stereotyped PortUsage Classes are shown within the frame of their container Stereotyped ComponentUsage Subsystem.

In the sample diagram below, the ProcessComponent used by the ComponentUsage, and the ProtocolPort used by the PortUsage is shown. The use and represents relationship is made explicit with the standard Generalization notation.

Note that some tools do not support the UML Subsystem as a first-class model element, but rather require the designer to use instances of Package, instead, and apply a "subsystem" Sterotype. A side effect of this workaround is that, as Package are not Classifier, some tools do not allow creation of Generalization relationships between the ProcessComponent Stereotype of Package (that should be Subsystem), and the ComponentUsage Stereotype of Package (that should be also Subsystem). In this case, an easy workaround is to use a Dependency, from the ComponentUsage to the ProcessComponent, and stereotype the dependency as 'uses'.

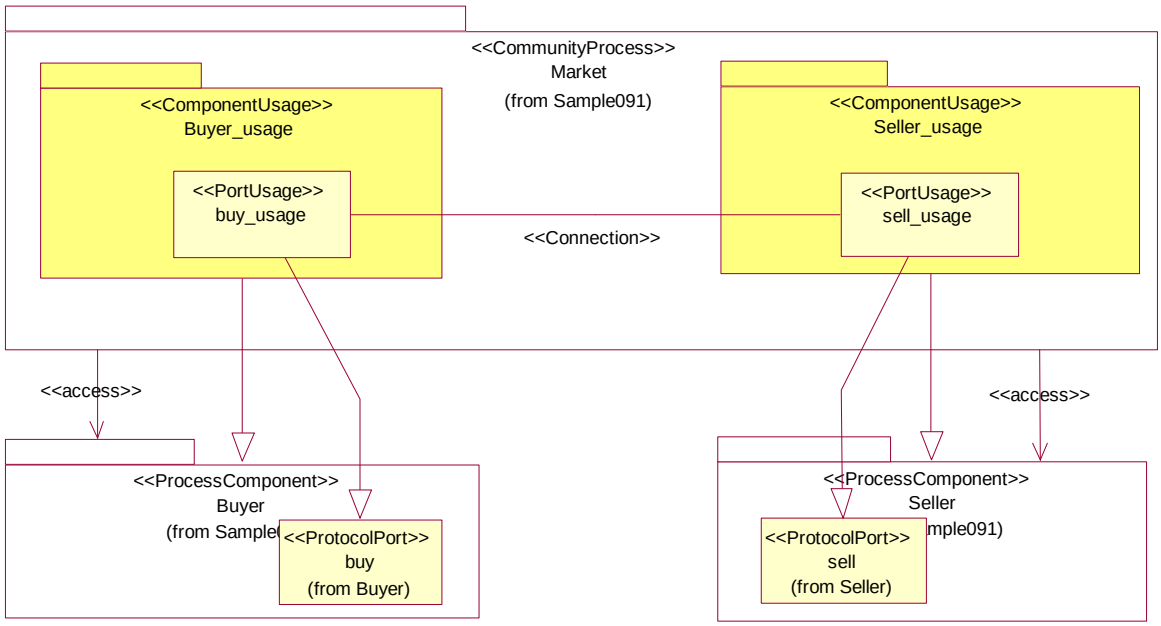


Figure 55: Sample Composition as a CommunityProcess (UML)

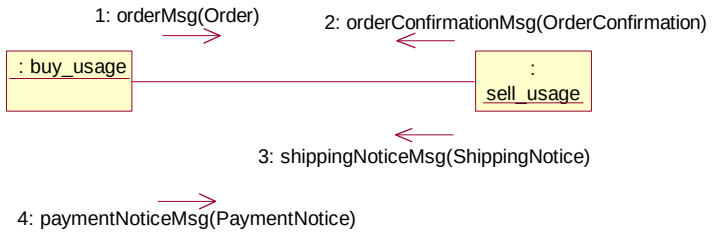


Figure 56: Sample Composition as a CommunityProcess, (UML Collaboration view)

7.2.4.2 ContextualBinding on Community Process

Representation of a ContextualBinding in UML uses the same artifacts and notation than the standard UML Binding.

Note that some tools have no direct support for the Binding three-way dependency. A workaround is to use a purely graphical note artifact, locate it within the frame of the Composition (or its container ComposedComponent) and reference from the note both the 'fills' ComponentUsage, and the 'bindsTo' ProcessComponent.

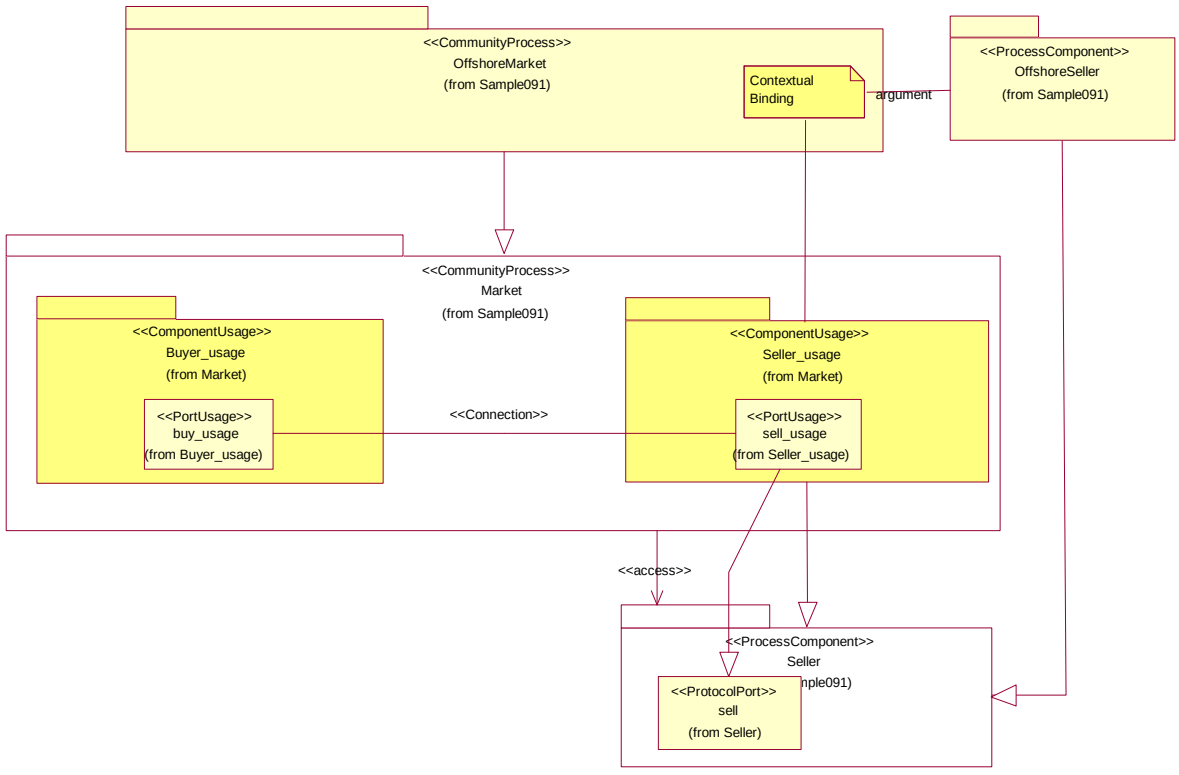


Figure 57: ContextualBinding on CommunityProcess (UML)

Alternatively, a more compact notation could use such a note as a compartment to textually express by name, the ContextualBinding of the 'bindsTo' ProcessComponent, to the 'fills' ComponentUsage. This is similar to the notational approach used by CCA, for representation of ContextualBinding.

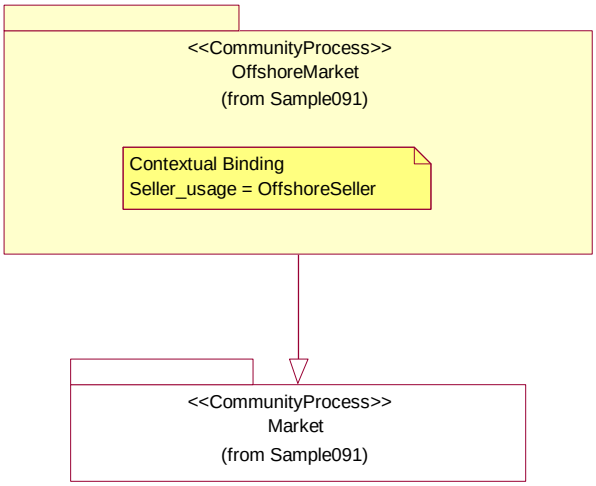


Figure 58: ContextualBinding on CommunityProcess, compact form (UML)

7.2.5 ComponentRealization examples

See also examples for Composition «profile» Package, section 7.1.4, page 121.

7.2.5.1 ComposedComponent

The UML representation of ComposedComponent in a standard Class diagram is a combination to the representation of ProcessComponent (see section 7.2.3 "ComponentSpecification examples", in page 135, above) and Composition (see section 7.2.4 "Composition examples" in page 136, above).

An additional PortProxy stereotyped Class is located within the frame of the ComposedComponent stereotyped Subsystem. A Generalization relationship is explicitly used in the diagram below, to express the ProtocolRole that the PortProxy realizes.

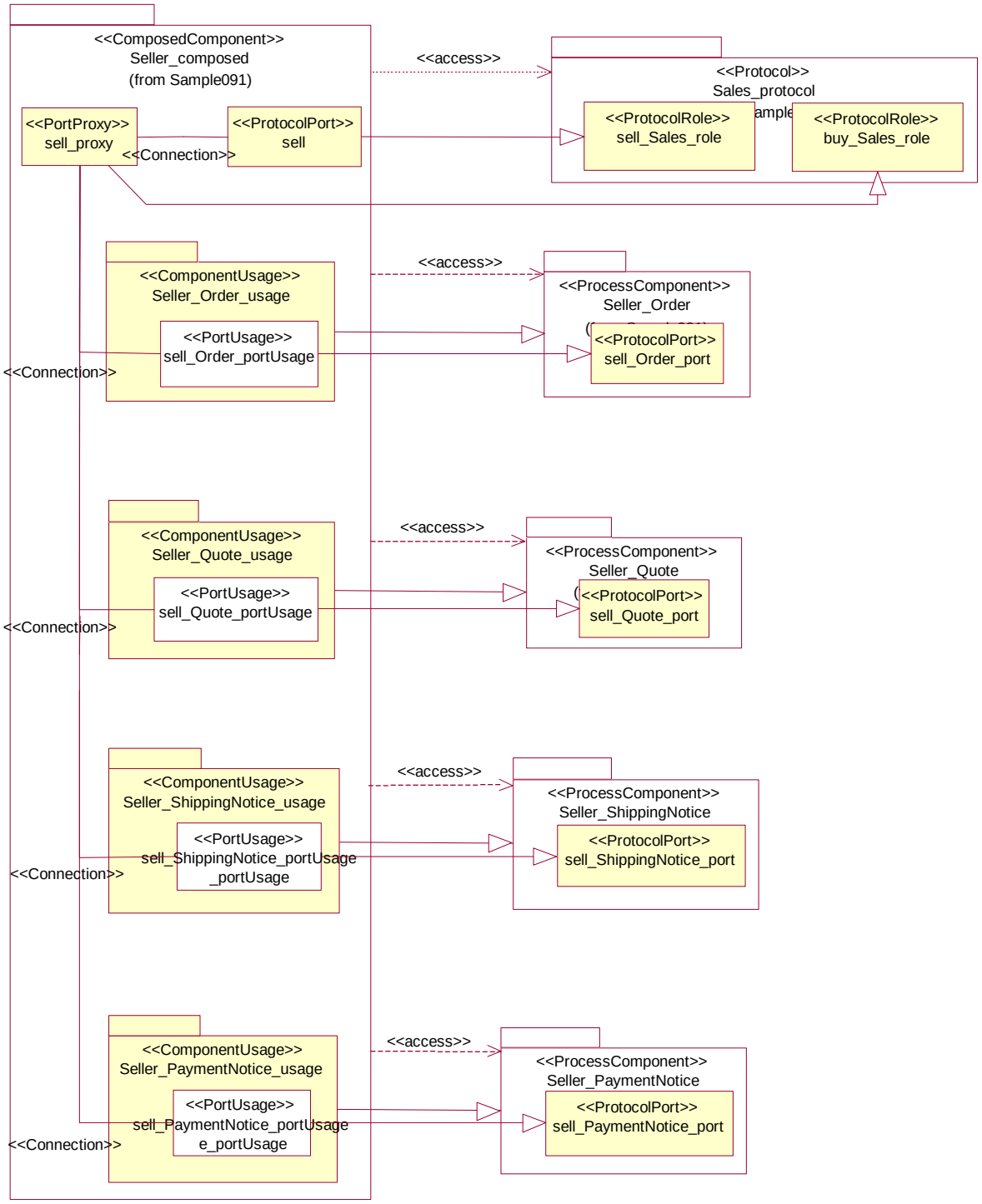


Figure 59: ComposedComponent (UML)

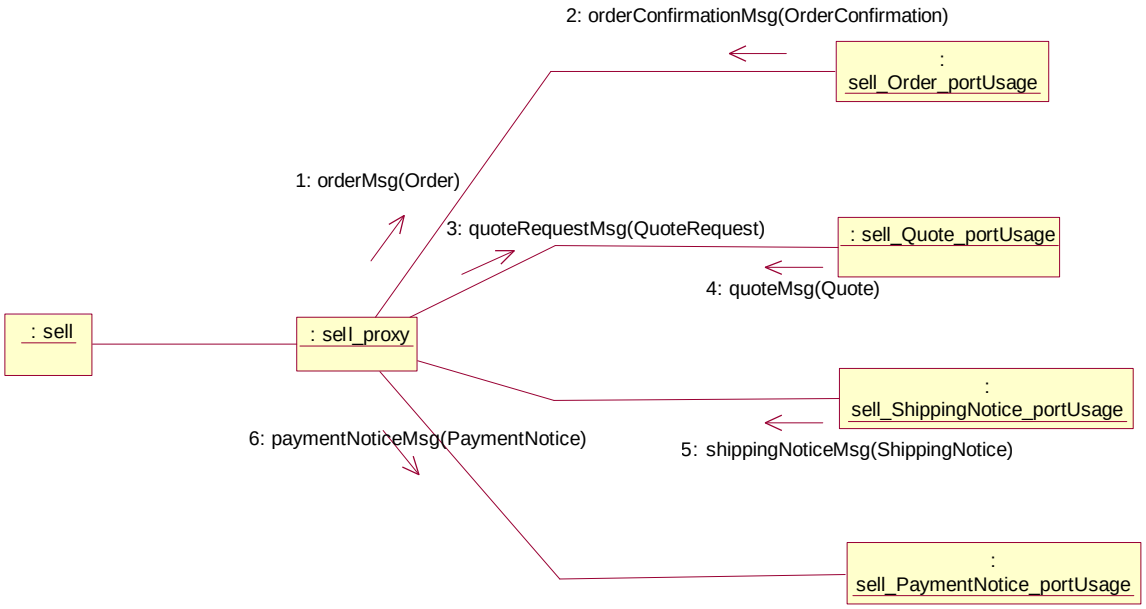


Figure 60: Composition of ComposedComponent (UML Collaboration view)

7.2.6 Choreography examples

7.2.6.1 Choreography of a Protocol

Samples of Choreograpy of Protocol and Process component have already been provided in UML examples section 7.2.2 "Protocol examples", in page 129. Other examples, in the similar (or identical in many cases) CCA notation have been provided in CCA examples section 7.1.2 "Protocol examples" in page 117.

7.3 UML-RT Notation

7.3.1 DocumentModel examples

UML utilizes for the specification of structural message payloads, the standard UML model elements and notation. Please refer to UML examples section 7.2.1 "DocumentModel examples" in page 128.

7.3.2 Protocol examples

7.3.2.1 Protocol,RequestReplyProtocol, FlowProtocol

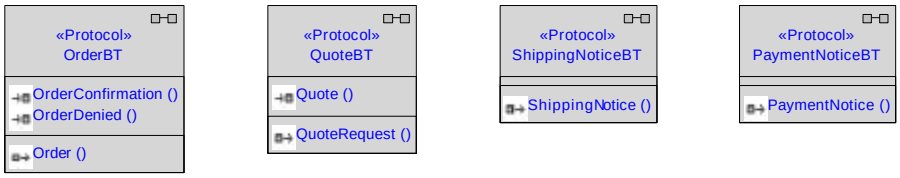


Figure 61: Sample Protocol, RequestReplyProtocol, FlowProtocol (RT)

7.3.2.2 Protocol with SubProtocols

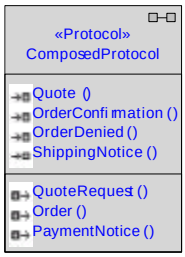


Figure 62: Sample Protocol with messages manual copied from SubProtocols (RT)

7.3.3 ComponentSpecification examples

7.3.3.1 ProcessComponents

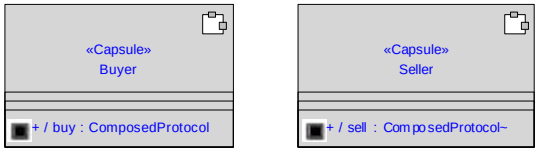


Figure 63: Sample ProcessComponents , Class view (RT)

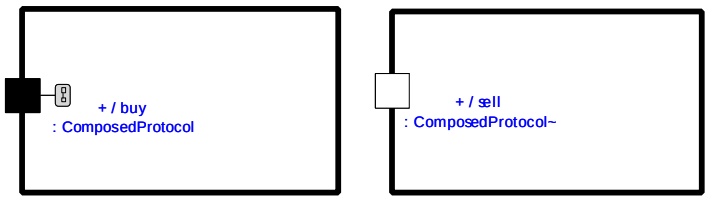


Figure 64: Buyer and Seller ProcessComponents , Buyer, Structure Diagrams (RT)

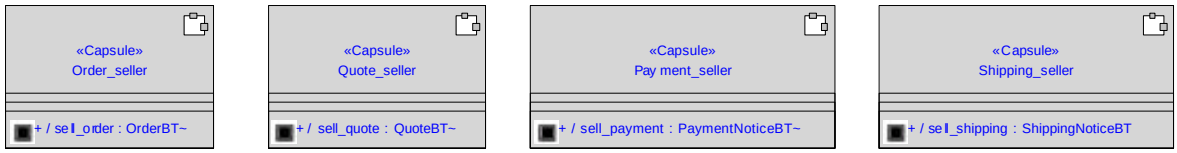


Figure 65: Some components for the ComposedComponent example, Class view (RT)

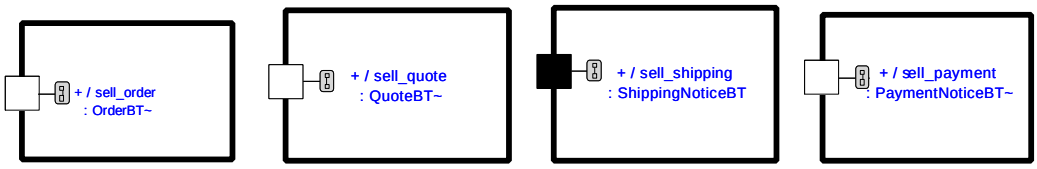


Figure 66: Order_seller, Quote_seller_Payment_seller, Shipping_seller: Structure Diagrams (RT)

7.3.4 Composition examples

7.3.4.1 Composition (as a CommunityProcess)

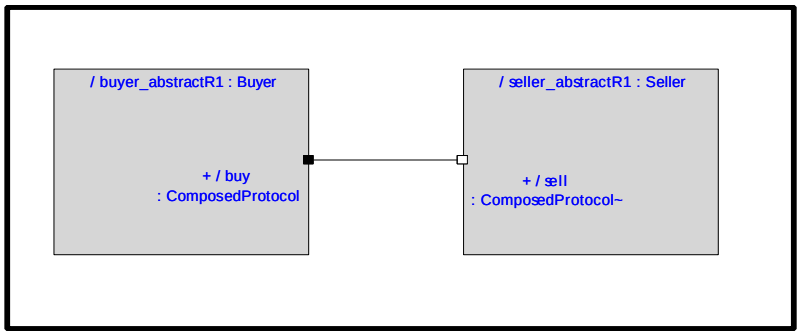


Figure 67: Sample Composition as a CommunityProcess. Structure Diagram (RT)

7.3.4.2 ContextualBinding on Community Process

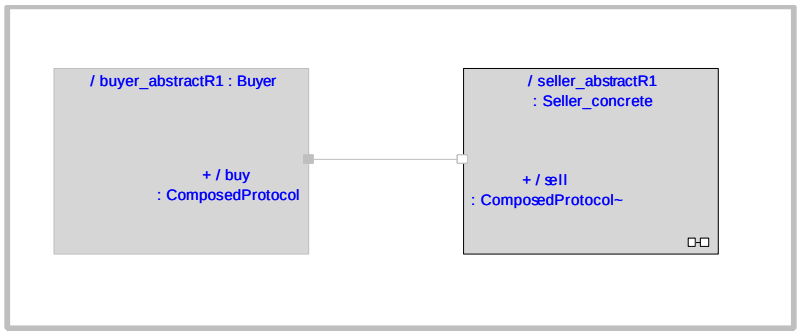


Figure 68: Specialized Composition (RT)

7.3.5 ComponentRealization examples

See also examples for Composition «profile» Package, section 7.1.4, page 121.

7.3.5.1 ComposedComponent

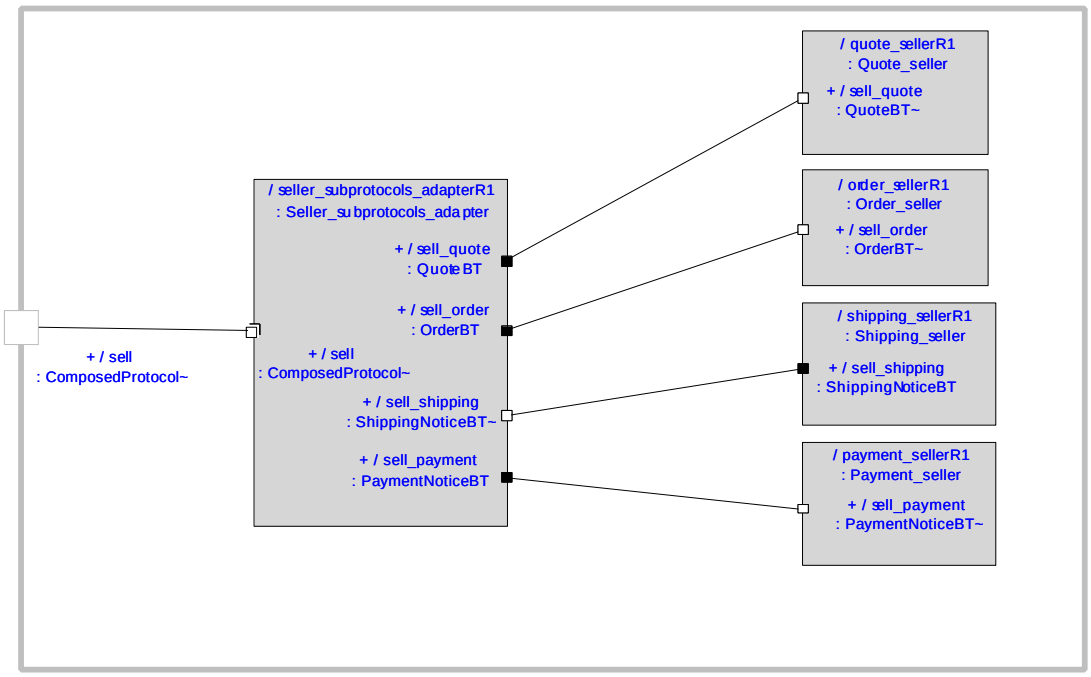


Figure 69: ComposedComponent (RT)

7.3.6 Choreography examples

7.3.6.1 Choreography of a Protocol

UML-RT utilizes the State Machine model elements, and StateChart notation, to specify the sequence of interactions in a protocol. Specifications similar to the ones referred in the CCA and UML example sections should be applicable in the UML-RT.

8. *Proof of correctness*

To prove that the Virtual metamodel can be used to construct model instances, properly expressing the concepts in the Conceptual Meta-Model, using the UML baseClasses and their relationships.

A number of collaboration diagrams, at the instance level, are presented below.

These are instances (M1) of the Virtual metamodel UML classes and stereotypes (M2).

The examples presented in Section 7 – "Samples" in page 116, are rendered here.

For each model element in the examples – whether classifier, relationship or feature- a box is included in the diagram. For each metarelationship between metamodel elements, a line is included in the diagram.

The author apologizes, if role names are difficult to read, or obscure parts of the diagram. The author was unable to re-position role name texts in the diagram, for improved legibility.

8.1.1 *DocumentModel proof*

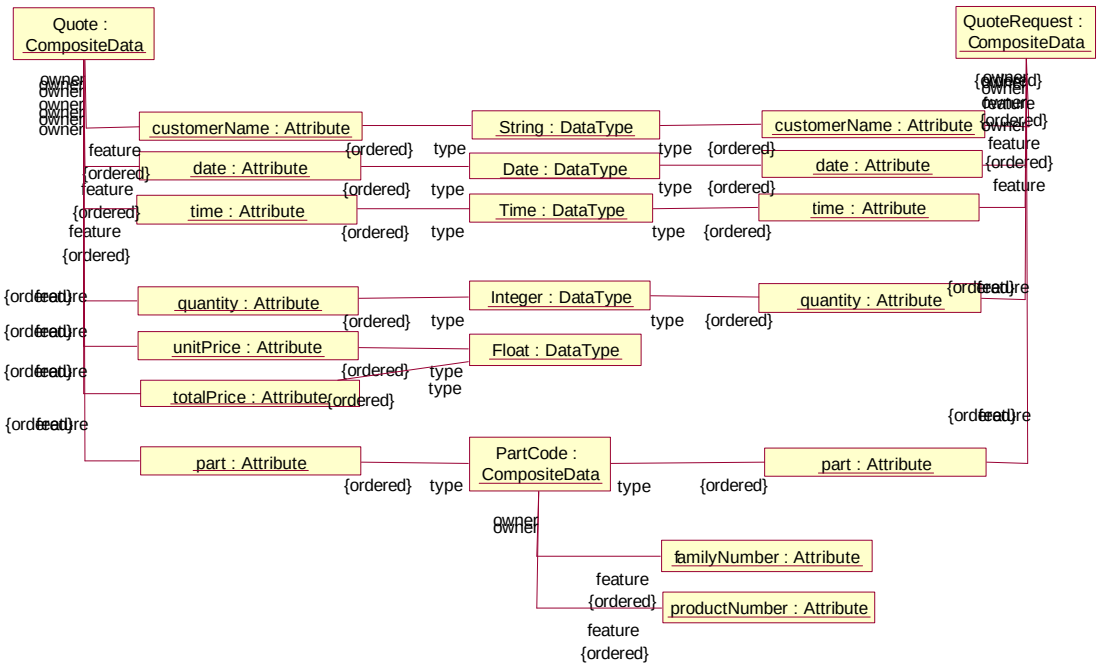


Figure 70: CompositeData (M1s)

8.1.2 Protocol proof

8.1.2.1 Protocol, RequestReplyProtocol, FlowProtocol

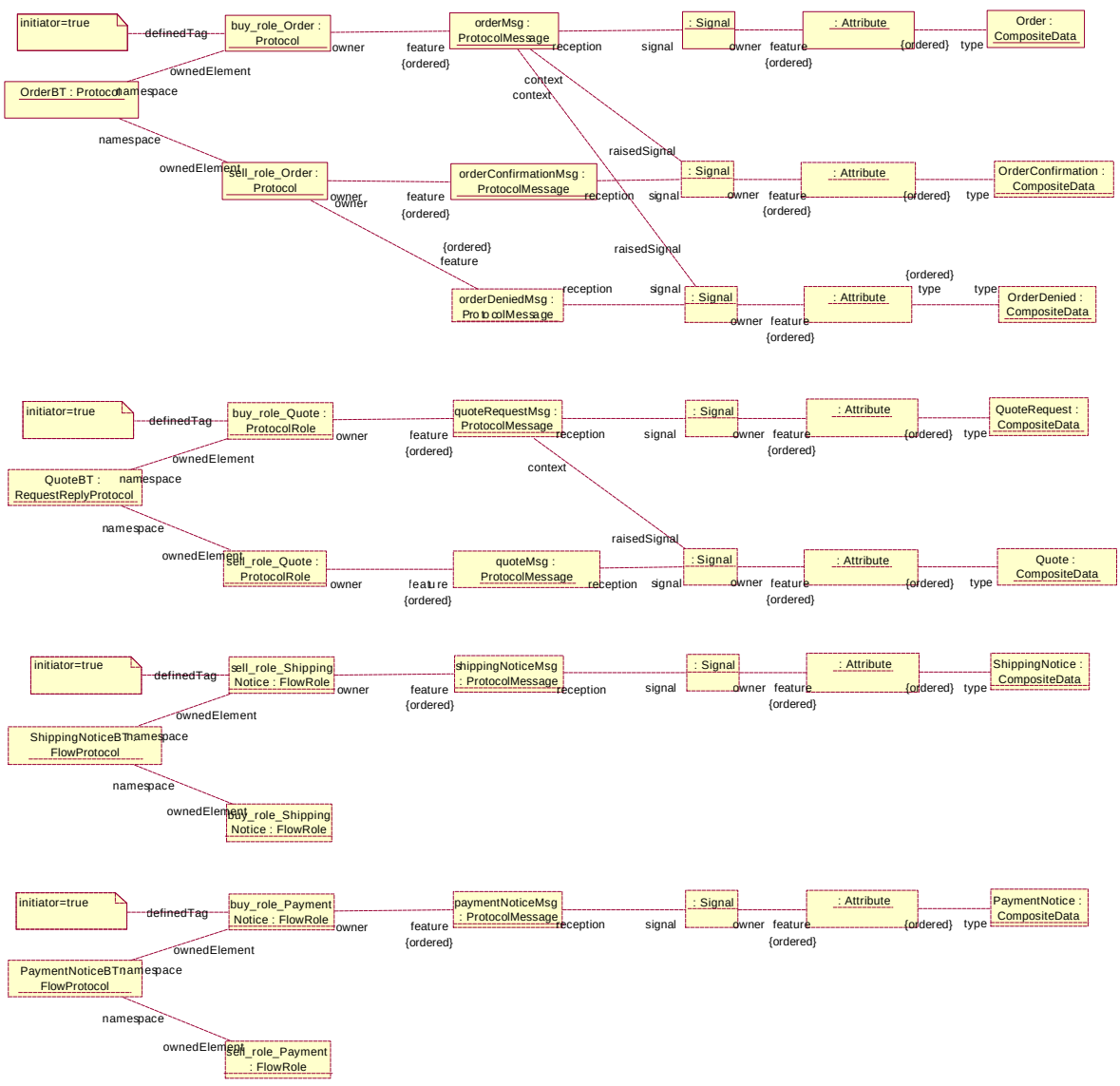


Figure 71: Sample Protocol, RequestReplyProtocol, FlowProtocol (M1s)

8.1.2.2 Protocol with SubProtocols

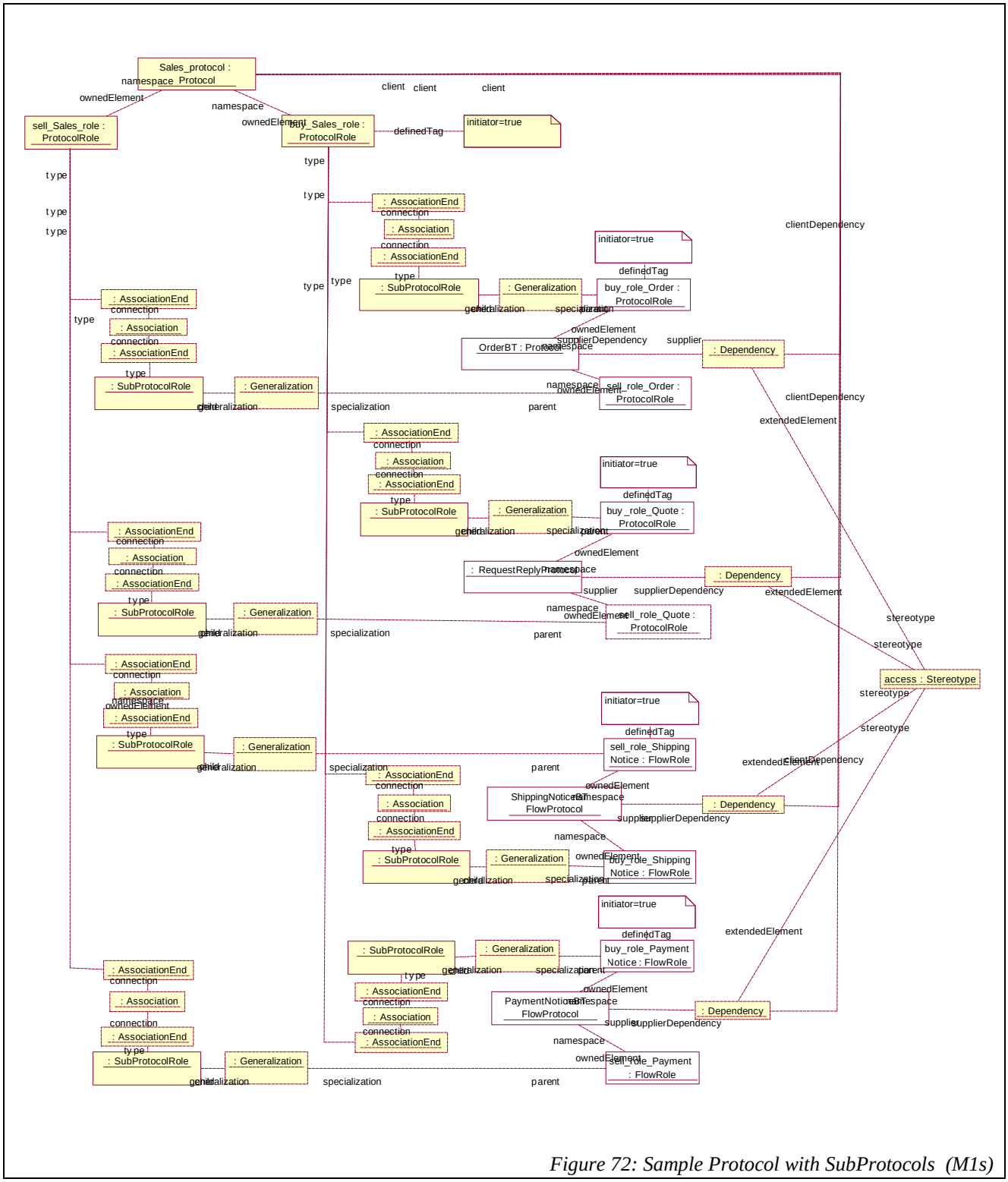


Figure 72: Sample Protocol with SubProtocols (M1s)

8.1.3 ComponentSpecification proof

8.1.3.1 ProcessComponents

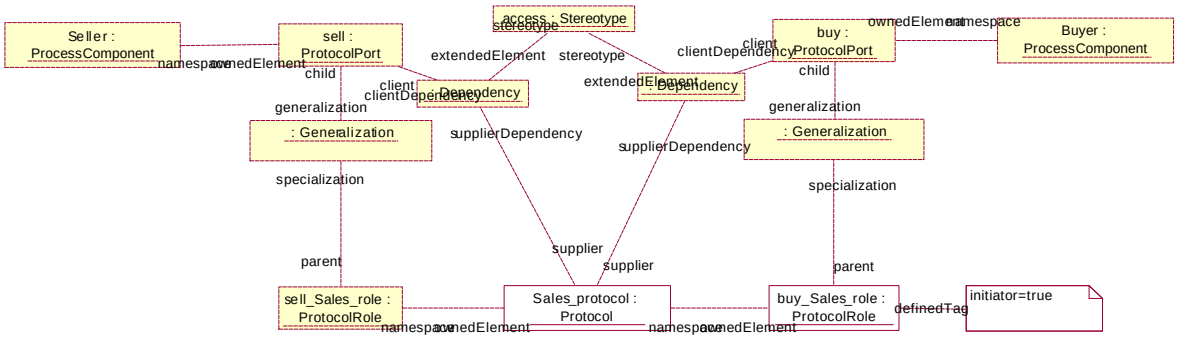


Figure 73: Sample ProcessComponents, with PropertyDefinitions, and ProtocolPorts (M1s)

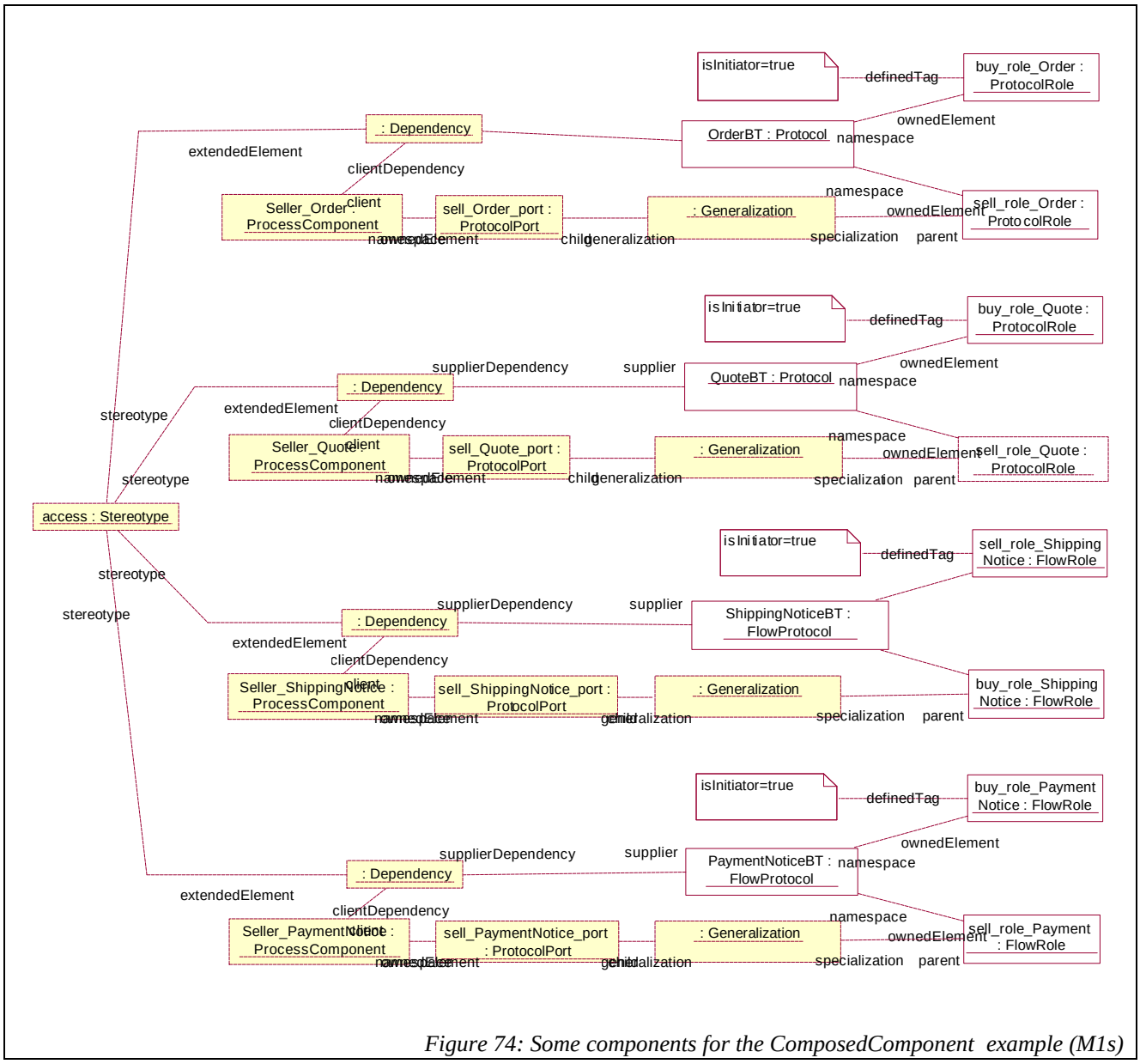


Figure 74: Some components for the ComposedComponent example (M1s)

8.1.4 Composition proof

8.1.4.1 Composition (as a CommunityProcess)

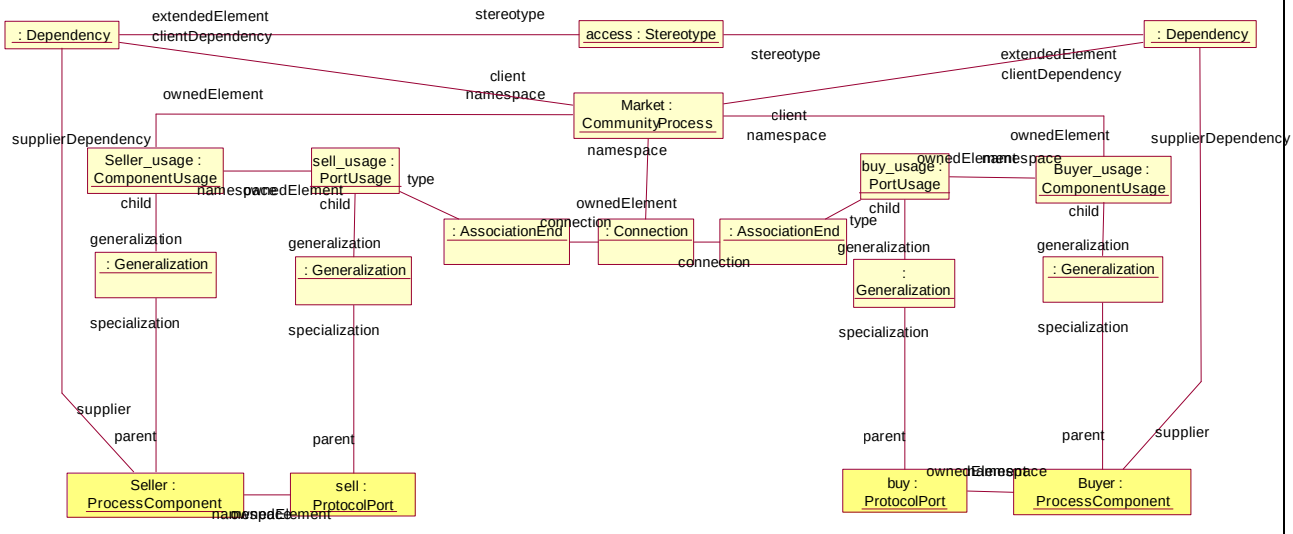


Figure 75: Sample Composition as a CommunityProcess (M1s)

8.1.4.2 ContextualBinding on Community Process

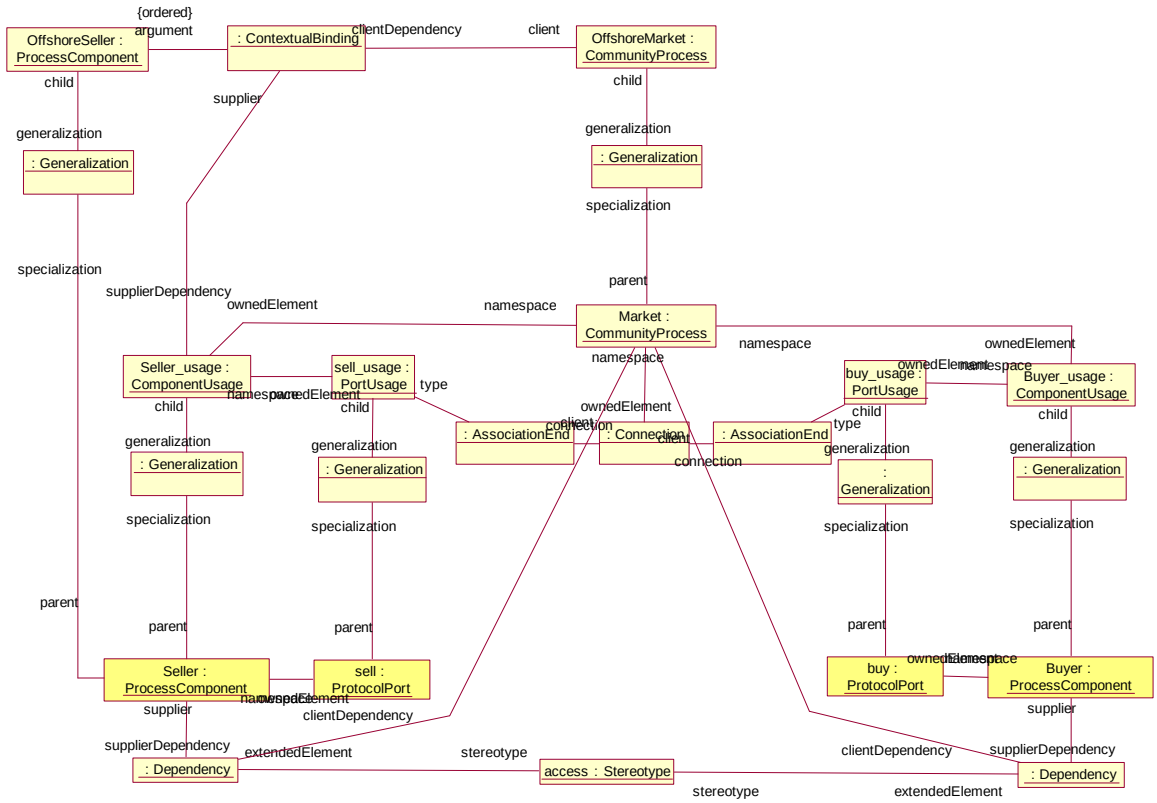


Figure 76: Specializing Composition with ContextualBinding (as a CommunityProcess) (M1s) (as Collaboration)

8.1.5 ComponentRealization proof

See also proof for Composition «profile» Package, section 8.1.4, page 150.

8.1.5.1 ComposedComponent

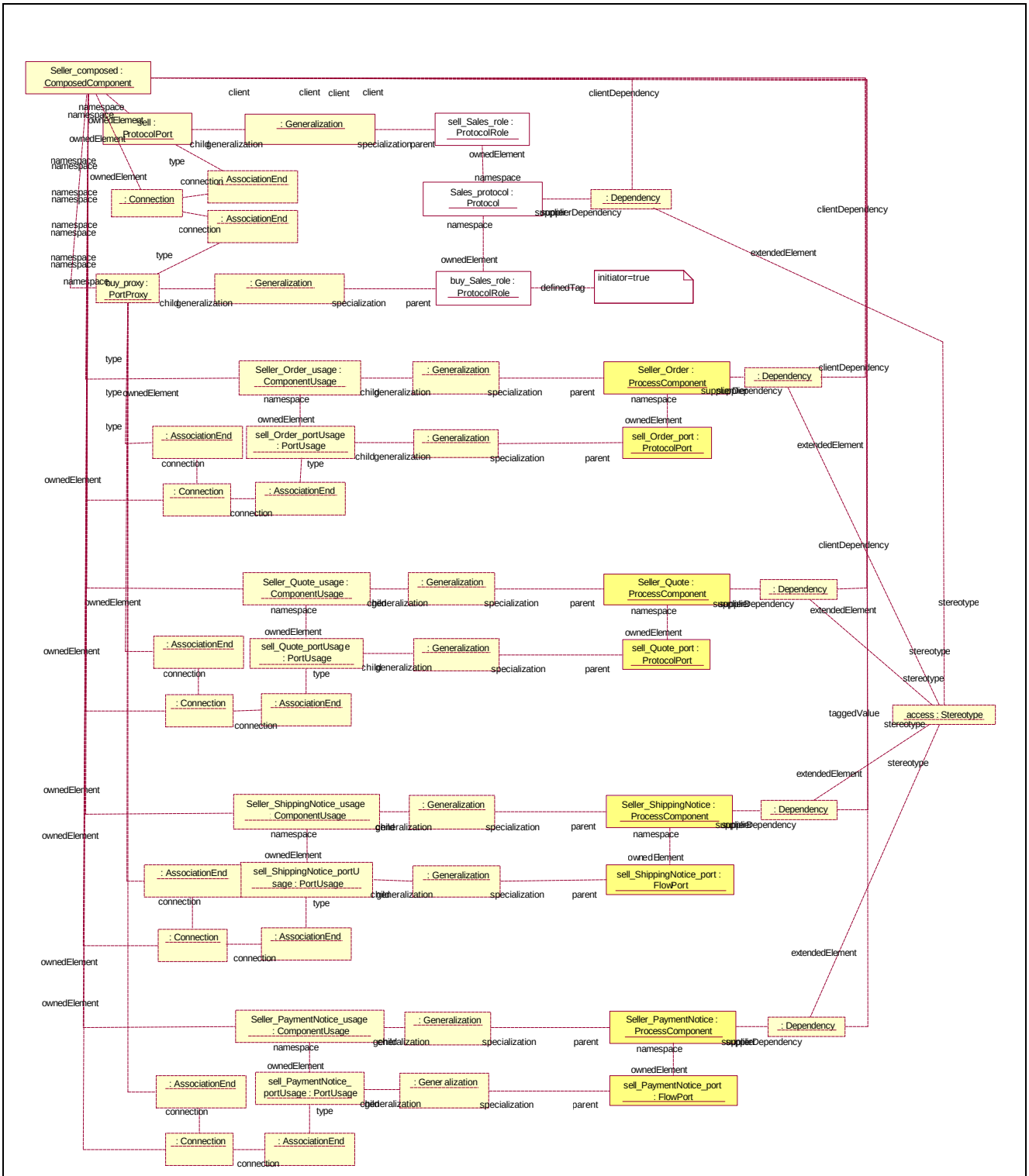


Figure 77: ComposedComponent (M1s)

8.1.6 Choreography proof

8.1.6.1 Choreography of a Protocol

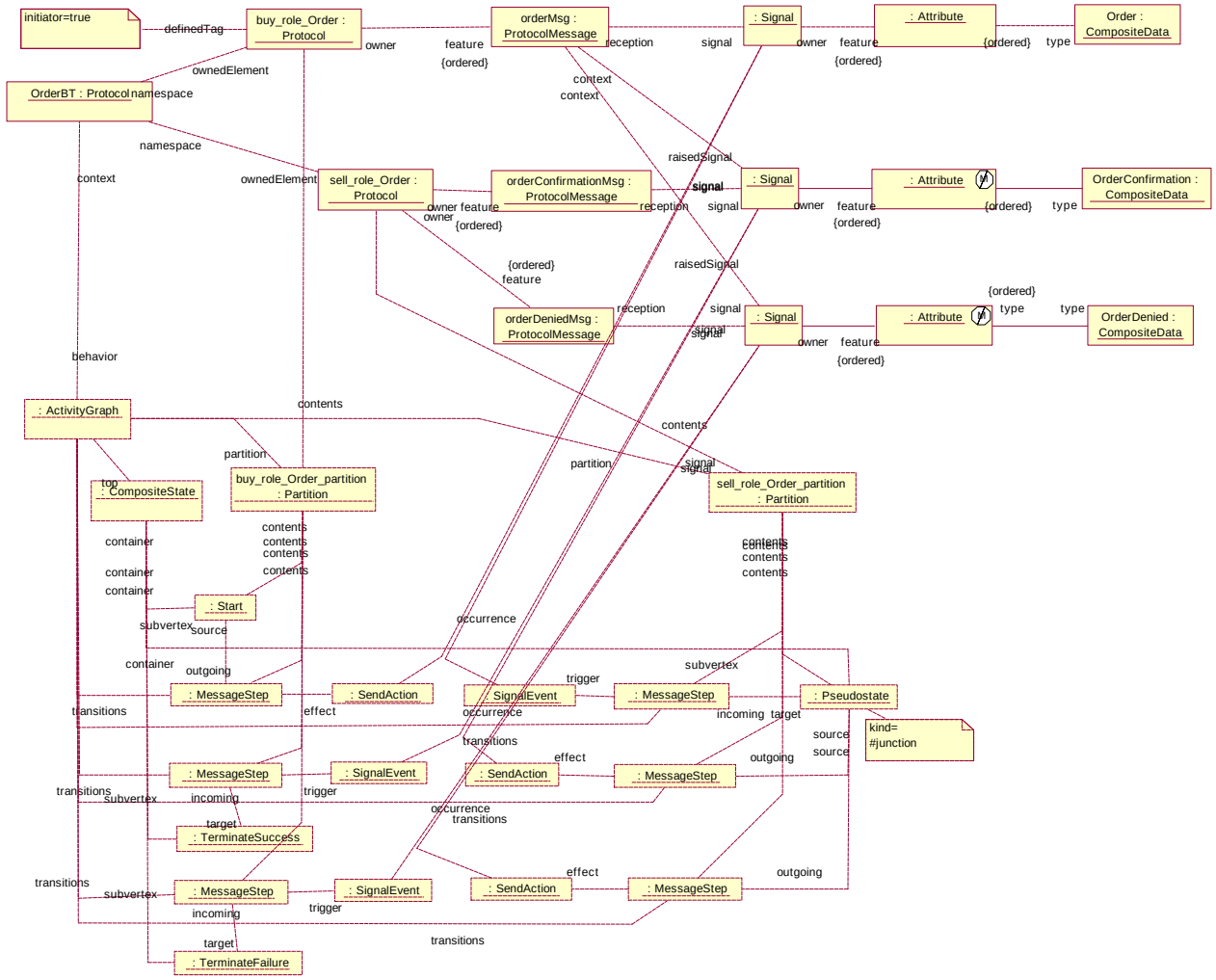


Figure 78: Choreography of a Protocol (M1s)

8.1.6.2 Choreography of a Protocol with sub-Protocols

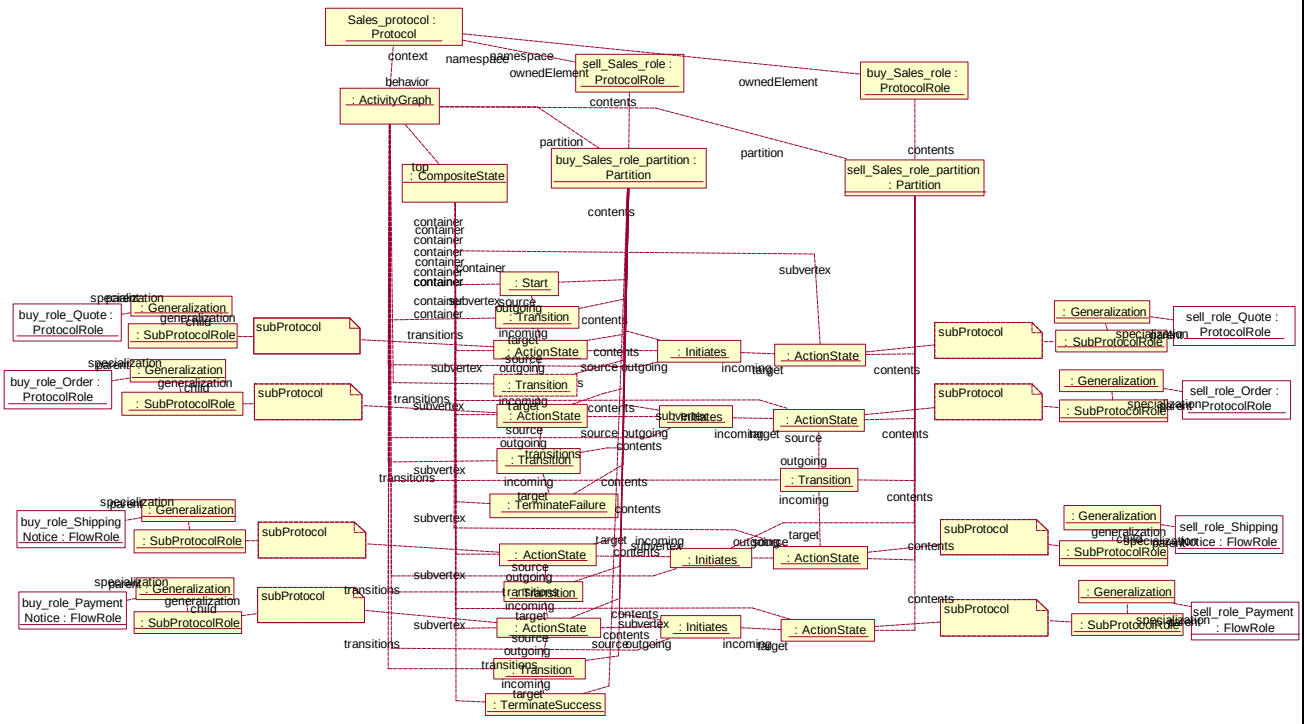


Figure 79: Choreography of a Protocol with sub-Protocols (M1s)

8.1.7 High Level Activity Graph of Composition proof

8.1.7.1 High Level Activity Graph of a Composition

Figure 80: High Level Activity Graph of a Composition (M1s)

9. References

[UML1.4] OMG Unified Modeling Language Specification, Version 1.4 beta R1, November 2000, OMG Document ad/ 2000-11-01
http://cgi.omg.org/cgi-bin/doc?ad/00-11-01.pdf

[OORAM] "Working with Objects : the OORAM Software Engineering Method", Trygve Reenskaugh, Per Wold and Odd Arild Lehne, 1996 Manning Publications Co. ISBN 1-884777-10-4 also by Prentice-Hall ISBN 0-13-452930-8

[ROOM] "Real-Time Object-Oriented Modeling", Bran Selic, Garth Gullekson and Paul T. Ward, 1994 John Willey & Sons, Inc. ISBN 0-471-59917-4

[UML-RT] "Using UML for Modeling Complex Real-Time Systems", Bran Selic, ObjectTime Limited, Jim Rumbaugh, Rational Software Corporation, March 11, 1998. http://www.rational.com/media/whitepapers/umlrt.pdf

[CATALYSIS] "Objects, Components and Frameworks with UML – The Catalysis^{SM} Approach" Desmond Francis D'Souza and Alan Cameron Willis, 1999 Addison-Wesley ISBN 0-201-31012-0