Enterprise Collaboration Architecture (ECA) Specification

February 2004
Version 1.0
formal/04-02-01

OBJECT MANAGEMENT GROUP

An Adopted Specification of the Object Management Group, Inc.

Contents

Contents

Contents

Preface

About the Object Management Group

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

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

Intended Audience and Use

The information described in this manual is aimed at managers and software designers who want to produce applications that comply with the family of OMG standards. The benefit of compliance is, in general, to be able to produce interoperable applications that run in heterogeneous, distributed environments.

Context of OMG Modeling

The OMG is dedicated to producing a framework and specifications for commercially available object-oriented environments. The Object Management Architecture (as defined in the *Object Management Architecture Guide*) is the umbrella architecture for OMG specifications. The defining model for the architecture is the Reference Model, which classifies the components, interfaces, and protocols that compose an object system. The Reference Model consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification*

- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification.*

- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

- **Application Objects**, which are objects specific to particular commercial products or end user systems. Application Objects correspond to the traditional notion of applications, so they are not standardized by the OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

- **OMG Modeling,** a collection of modeling specifications that advance the state of the industry by enabling OO visual modeling tool interoperability. OMG Modeling provides a set of CORBA interfaces that can be used to define and manipulate a set of interoperable metamodels.

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc., at:

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

Typographical Conventions

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

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

`Courier bold` - Programming language elements.

Helvetica - Exceptions

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

Acknowledgments

This specification was prepared by the following companies:

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

Supporting companies are:

- Adaptive
- Hitachi
- Netaccount
- SINTEF

Introduction 1

Contents

This chapter includes the following topics.

1.1 Guide to the Specification

1.1.1 Overall Structure of the Specification

Chapter 1 introduces the specification.

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

- The analysis phase when the roles played by the system's components in the business it supports are defined and related to the business requirements.
- The design and implementation phases, when detailed specifications for the system's components are developed.

- The maintenance phase, when, after implementation, the system's structure or behavior is modified and tuned to meet the changing business environment in which it will work.

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

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

1.2 Conformance Issues

1.2.1 Summary of optional versus mandatory interfaces

For a modeling tool to claim compliance to the EDOC specification it must implement at least one of the mandatory compliance points in Section 1.2.2.1, "Mandatory Compliance Points," on page 1-3, and state the name of the compliance point(s). The mandatory compliance points are all variations on the ability to model or interchange designs using the Enterprise Component Architecture (ECA), which forms the core of EDOC.

There are a number of other normative profiles and metamodels contained within this specification, and these are given named optional compliance points in Section 1.2.3, "Optional Compliance Points," on page 1-3.

1.2.2 Compliance Points

1.2.2.1 Mandatory Compliance Points

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

Table 1-1 Mandatory Compliance Points

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

The columns in Table 1-1 are defined as follows:

MOF Repository

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

MOF XMI interchange

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

1.2.3 Optional Compliance Points

The specification has the following optional compliance points:

CCA Notation

1.3 Proof of Concept

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

1.3.1 Data Access Technologies

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

1.3.2 DSTC

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

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

1.3.3 EDS

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

1.3.4 Fujitsu

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

1.3.5 IBM

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

1.3.6 Iona

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

1.3.7 Open-IT and SINTEF

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

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

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

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

1.3.8 Sun Microsystems

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

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

1.3.9 ebXML

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

ECA: Rationale and Application 2

Contents

This chapter includes the following topics.

Section I - Vision

2.1 Overview

The vision of the Enterprise Collaboration Architecture is to simplify the development of component based EDOC systems by means of a modeling framework and conforming to the OMG Model Driven Architecture (see [30] in Appendix A), that provides:

- A platform independent, recursive collaboration based modeling approach that can be used at different levels of granularity and different degrees of coupling, for both business and systems modeling and encompasses:
  - A loosely coupled, re-usable business collaboration architecture that can be leveraged by business-to-business (b2b) and business-to-customer (b2c) applications, as well as for enterprise application integration.
  - A business component architecture that provides interoperable business components and services, re-use and composability of components and re-use of designs and patterns, while being independent of choice of technology (e.g., component models), independent of choice of middleware (e.g., message services) and independent of choice of paradigms (e.g., synchronous or asynchronous interactions).
- Modeling concepts for describing clearly the business processes and associated rules that the systems support, the application structure and use of infrastructure services, and the breakdown of the system into configurable components.
- An architectural approach that allows the integration of “process models” and “information models.”
- A development approach that allows two-way traceability between the specification, implementation and operation of Enterprise computing systems and the business functions that they are designed to support.
- Support for system evolution and the specification of collaboration between systems.
- A notation that is accessible and coherent.

The vision addresses key business needs by enabling the development of tools that support:

- Business collaborations as a central concern – covering alliances, outsourcing, supply chains, and internet commerce, and dealing with relationships that are in constant flux where what is inside the enterprise today is outside tomorrow, and vice versa.
- Process engineering by assembling services – so that basic business functions can remain relatively constant while who performs them and in what sequence changes, and services themselves can become proactive.
- The ability for parts of the enterprise to react quickly and reliably to change through:

- Shorter development time and improved quality of applications meeting market needs, improved interoperability between systems and support for distributed computing.
- Reduced lead time and improved quality resulting from the ability to generate a substantial portion of application code.
- More robust specification by removing ambiguity and enabling more rigorous analysis of designs.

- A new marketplace for interoperable collaboration based infrastructures and business components.

This remainder of this chapter:

- provides an overview of the ECA (Section II), and
- defines how the ECA and other specifications are applied in the specification of an EDOC system (Section III).

Section II - The ECA Elements

2.2 The Enterprise Collaboration Architecture

The Enterprise Collaboration Architecture (ECA) comprises the following set of UML models:

- The Component Collaboration Architecture (CCA) which details how to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system.
- The Entities model, which describes how to model entity objects that are representations of concepts in the application problem domain and define them as composable components.
- The Events model, which describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model event driven systems.
- The Business Process model, which specializes the CCA, and describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model system behavior in the context of the business it supports.

The semantics of each model are expressed in a UML-independent MOF metamodel.

The ECA models are technology independent and are used together to define platform independent models of EDOC systems in conformance with the MDA. In particular, they enable the modeling of the concepts that until now have had to be specified programmatically in terms of the use of services such as events/ notifications, support for relationships and persistence.

2.2.1 Component Collaboration Architecture

The Component Collaboration Architecture (CCA) details how to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system. It defines an architecture of recursive decomposition and assembly of parts, which may be applied to many domains.

The term component is used here to designate a logical concept - a "part," something that can be incorporated in a logical composition. It is referred to in the CCA as a Process Component. In many cases Process Components will correspond, and have a mapping, to physical components and/or deployment units in a particular technology.

A Process Component is a processing component: it collaborates with other Process Components within a CCA Composition, interacting with them through Ports, where Ports are an abstraction of interfaces of various types (e.g., synchronous, asynchronous). Process Components can be used to build other Process Components or to implement roles in a process – such as a vendor in a buy-sell process.

Process Components collaborate at a given level of specification collaborate and are themselves decomposed at the next lower level of specification. Thus the concepts of Process Component and Composition are interdependent.

The recursive decomposition of Process Components utilizes two constructs in parallel: Composition to show what Process Components must be assembled and how they are put together to achieve the goal, and Choreography to show the flow of activities to achieve a goal. The CCA integrates these concepts of "what" and "when" at each level.

Since CCA, by its very nature, may be applied at many levels and the specification requirements at these various levels are not exactly the same, the CCA can be further specialized with models for each level using the same model mechanisms. Thus 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, and there may be requirements for Process Components 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 and these can decompose into program level Process Components within a single system. However, this capability for recursive decomposition is not always required. Any Process Component may be implemented directly in the technology of choice without requiring decomposition into other Process Components.

2.2.2 Entities Model

The Entities model describes a set of model elements that may be used to model entity objects that are representations of concepts in the application problem domain and define them as composable components.

The goal is to define the entities with their attributes, relationships, operations, constraints, and dependencies at a technology-independent level as components within a system modeled using the CCA. The component determines the unit of distribution and interfaces that must be complemented by other components. The model includes declarative elements for placing constraints on the model and for rules that will propagate the effects of changes and events.

The Entities model is used with the Events and Business Process models to allow definition of the logic of automated business processes and of events that may be exchanged to achieve more loosely coupled integration. These three models together support the design of an EDOC system on the foundation provided by the CCA.

The Entities model is used to define a representation of the business and operations that effect changes in state of the business model. Business processes modeled using the Events model and the Business Process model operate on this model where the process flow determines when operations should occur as a result of inputs from other systems, the occurrence of business events, or the actions of human participants.

2.2.3 Events Model

The Events model describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model event driven systems.

An event driven system is a system in which actions result from business events. Whenever a business event happens anywhere in the enterprise, some person or thing, somewhere, may react to it by taking some action. Business rules determine what event leads to what action. Usually the action is a business activity that changes the state of one or more business entities. Any state change to a business entity may constitute a new business event, to which, in turn, some other person or thing, somewhere else, may react by taking some action. The purpose of the Event Model is to define the use of the concepts in the CCA, Entity and Event models, and to extend them in order to support the design of event-driven business systems.

The main concepts in event driven business models are the business entity, business event, business process, business activity, and business rule. So the basic building blocks are the business process and the business entity. The two are 'wired together' by a flow of actions from process to entity, and by a flow of events from entity to process. In a component framework, therefore, business processes have event inflow and action outflow, and entities have action inflow and event outflow.

This means that CCA business process components and CCA business entity components can be created by modeling:

- A business process as a set of rules of the type notification/condition/activity (This is the event-driven equivalent of the commonly known event/condition/action rule).
- A business entity as set of operation/state/event causalities.

The connection from business process to business entity is a configurable mapping of activity to operation.

The connection from business entity to business process is a configurable set of subscriptions.

With these building blocks it is possible to model a number of event-based interactions. Furthermore, by reconfiguring the activity to operation mapping and/or the subscriptions, it is possible to re-engineer the business process and its execution in the system.

However, neither the business world, nor the computing world applies only one paradigm to their problem space. Businesses use a combination of loosely coupled and tightly coupled processes and computing solutions deploy a combination of loosely coupled and tightly coupled styles of communication and interaction between distributed components. Consequently, while the Events model is defined to support the event-driven flavor of loosely coupled business and systems models, it allows such models to co-habit with more tightly coupled models.

2.2.4 Business Process Model

The Business Process model specializes the CCA, and describes a set of model elements that may be used on their own, or in combination with the other ECA elements, to model system behavior in the context of the business it supports.

The Business Process model provides modeling concepts that allow the description of business processes in terms of a composition of business activities, selection criteria for the entities that carry out these activities, and their communication and coordination. In particular, the Business Process model provides the ability to express:

- Complex dependencies between individual business tasks (i.e., logical units of work) constituting a business process, as well as rich concurrency semantics.
- Representation of several business tasks at one level of abstraction as a single business task at a higher level of abstraction and precisely defining relationships between such tasks, covering activation and termination semantics for these tasks.
- Representation of iteration in business tasks.
- Various time expressions, such as duration of a task and support for expression of deadlines.
- Support for the detection of unexpected occurrences while performing business tasks that need to be acted upon (i.e., exceptional situations).
- Associations between the specifications of business tasks and business roles that perform these tasks and also those roles that are needed for task execution.
- Initiation of specific tasks in response to the occurrence of business events.
- The exposure of actions that take place during a business process as business events.

Section III - Application of the ECA Elements

2.3 Separation of Concerns and Viewpoint Specifications

The RFP states that:

"Successful implementation of an enterprise computing system requires the operation of the system to be directly related to the business processes it supports. A good object-oriented model for an enterprise computing system must therefore provide a clear connection back to the business processes and business domain that are the basis for the requirements of the system. However, this model must also be carried forward into an effective implementation architecture for the system. This is not trivial because of the demanding nature of the target enterprise distributed computing environment."[1]

This is reflected in the vision for this ECA to provide:

- A development approach that allows two-way traceability between the specification, implementation and operation of Enterprise computing systems and the business functions that they are designed to support.
- In order to clearly and coherently address these requirements, the specification of an EDOC system must be structured to address a number of distinct sets of concerns:
  - The behavior of the system, in the context of the business for which it is implemented (i.e., its roles in some enterprise that is greater than it itself), has to be specified in a way that can be traceably linked to its design.
  - The structure of the application processing carried out by the system has to be defined in terms of configurations of objects and the interactions between them.
  - The semantics of the application processing carried out by the system have to be expressed in a way that can be traceably linked from its roles through to the functions the system provides.
  - The infrastructure of the system has to be defined in terms of the use of object services to support the application processing structure.
  - The qualitative aspects of the system (e.g., performance and reliability objectives) have to be defined together with the hardware and software products that realize the system. These determine the physical configuration of application processing and supporting services across available resources, and how the system is managed.

This is the problem addressed by the Reference Model of Open Distributed Processing (RM-ODP) (see [1], [2], [3] Appendix A) and this specification uses as the conceptual framework for an EDOC system specification the concept of viewpoints defined in the RM-ODP. It partitions a system specification into five viewpoint specifications, namely the

- enterprise specification,
- computational specification,
- information specification,
- engineering specification, and
- technology specification.

1. RFP p19 under the heading of "Enterprise Computing Systems"

The set of linked specifications, taken together, ensure that the system can be implemented and operated in such a way that its behavior will meet the business needs of its owners, and, furthermore, that its owners will understand the constraints on their business that operation of the system will impose.

This section explains how the concepts defined by the ECA can be used to develop a full set of viewpoint specifications for an EDOC system and how specification integrity across the various viewpoint specifications can be ensured. In summary (Figure 2-1):

- The CCA, the Events model, the Entities model the Processes model and the Relationships model from the ECA are used, with relevant Patterns, to produce an *enterprise specification (Enterprise viewpoint)*.
- The CCA, the Entities model and the Events model from the ECA are used, with relevant Patterns, to produce a *computational specification (Computational viewpoint)*.
- The Entities model and Relationships model from the ECA are used, with relevant Patterns, to produce an *information specification (Information viewpoint)*.
- A technology abstraction model such as the Flow Composition Model (FCM), with relevant Patterns, is used to produce an *engineering specification (Engineering viewpoint)*.
- The mappings to various technologies, in particular, to J2EE with EJB, to CORBA 3 with CCM and to MS DNA/.NET with DCOM, are used to produce technology specifications (*Technology viewpoint*).

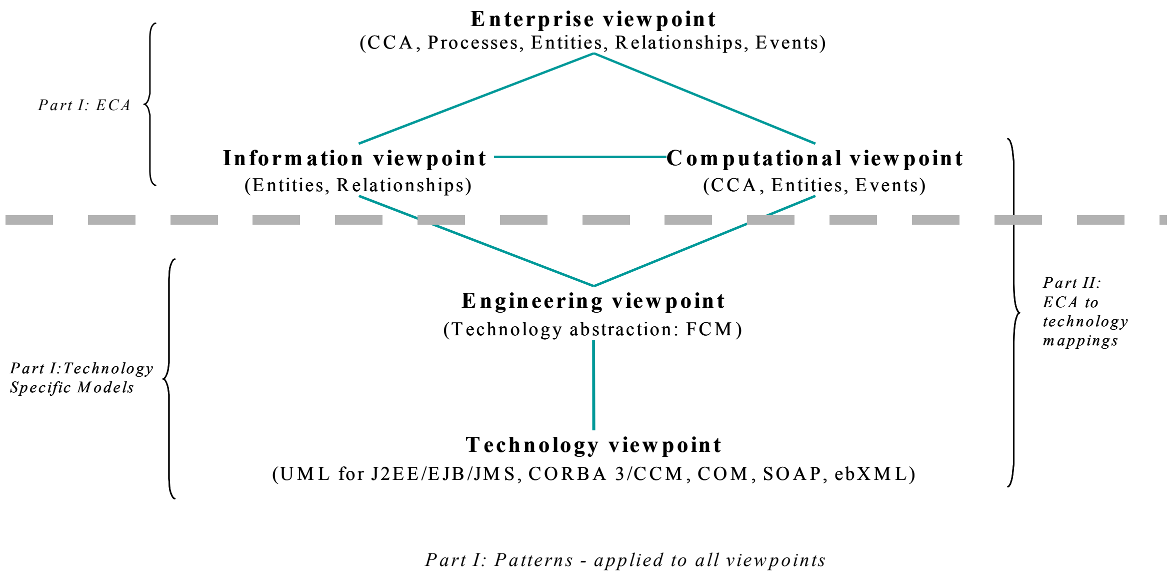

Enterprise viewpoint
(CCA, Processes, Entities, Relationships, Events)
Part I: ECA
Information viewpoint
(Entities, Relationships)
Computational viewpoint
(CCA, Entities, Events)
Engineering viewpoint
(Technology abstraction: FCM)
Part I:Technology Specific Models
Part II: ECA to technology mappings
Technology viewpoint
(UML for J2EE/EJB/JMS, CORBA 3/CCM, COM, SOAP, ebXML)
Part I: Patterns - applied to all viewpoints

Figure 2-1 ECA elements related to the ISO RM ODP viewpoints

Such a specification structure is valid for 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.
- Design and implementation phases, when detailed specifications for the system's components are developed.
- Maintenance phase, when, after implementation, the system's behavior is modified and tuned to meet the changing business environment in which it will work.

The overall structure of the ECA in the context of the ISO RM-ODP viewpoints is illustrated in Figure 2-1.

2.4 Enterprise Specification

2.4.1 Concepts

The enterprise specification of an EDOCsystem provides the essential traceability between the system design and the business processes and the business domain that are the basis for the requirement for the system.

The basis of the enterprise specification is provided by the concepts of the ODP enterprise language (modeled using the ECA elements). These concepts are defined in Appendix A - [4].

An enterprise specification models the structure and behavior of the system in the context of the business organization of which it forms a part in the following terms:

- the business processes supported by the system,
- steps in those processes and relationships between steps,
- business rules (policies) that apply to the steps,
- artifacts acted on by each step,
- enterprise objects representing the business entities involved,
- the roles that they fulfil in supporting the business processes, and
- the relationships between roles (including interaction relationships) where roles identify responsibility for steps in the business processes.

An EDOC system or each component of that system is modeled as an enterprise object and is assigned a role or roles in the community: hence, it is associated with specific parts of one or more processes. These roles identify the parts of the business processes for which the system is responsible and the artifacts that are involved. Such artifacts and resources represent the information held and acted upon by the system.

The central concept of any enterprise specification is that of a *community* that models a collection of entities interacting to achieve some purpose, which is defined by the *objective* of the community concerned. Each community is modeled as a configuration of *enterprise objects in roles*. The EDOC system of concern (or the components of that system) is modeled as one or more of the enterprise objects that are the members of the community.

The *behavior* of the members of the community is identified by the *roles* they fulfill, and is defined in terms of a set of *actions*, each of which may also be modeled as a *step* of one or more *processes*. Each process is designed to achieve the objective of the community.

Depending upon what it models, an enterprise object may be further refined as a community in a process of recursive decomposition.

Policies (business rules) may be associated with any other enterprise language concept and may be expressed in the form of constraints on any concept, or relationship between two concepts.

2.4.2 EDOC Enterprise Submodel

The EDOC enterprise specification makes use of the CCA for the role-based definition of the enterprise structure, where:

- Communities are modeled as Composed Components with associated Composition and Choreography definitions.
- Enterprise objects are modeled as ProcessComponents.
- The interactions in which enterprise objects can participate are defined by Ports and the associated Protocols.

It makes use of the Processes model for the process-based definition of the enterprise structure.

It makes use of the Event model for the definition of event driven enterprise structures.

It makes use of the Entities model for the definition of entities and rules. Artifacts, performers and responsible parties, which are the subject of the interactions, are modeled as entities.

It makes use of the Relationships model for rigorous specification of relationships.

2.5 Computational Specification

2.5.1 Concepts

The computational specification describes the implementation of the EDOC system (or components that comprise that system) in order to carry out the processing required by the system roles in the enterprise specification. It does this in terms of functional decomposition of the system into computational objects that interact at interfaces, and thereby enables distribution. It defines:

- Computational objects that play some functional role in the system and which can be described in terms of provided interfaces and used interfaces: a set of computational objects will correspond to the implementation of roles of the system in enterprise processes, and associated enterprise events and business rules.

- The interfaces at which the computational objects interact: this includes different types of interfaces and also describes data involved in computational interactions corresponding to the information objects in the information specification.
- The collaboration structures among a set of computational objects.

The computational viewpoint is closely related to the enterprise viewpoint in that the computational objects represent a functional mapping of enterprise concepts like business processes, rules, events etc. where these relate to the roles of the system in the enterprise specification. Ways of ensuring consistency (conformance/reference points) between enterprise and computational specifications should be supported (consistency statements for corresponding conformance/reference points in the two viewpoint specifications).

The EDOC computational specification concepts are based on the RM-ODP Part 3 Clause 7 (see Appendix A [3]).

2.5.2 EDOC Computational Specifications

An EDOC computational specification makes use of the CCA for the basic definition of the computational structure, where:

- Computational objects are modeled as ProcessComponents.
- The interfaces at which computational objects interact are modeled by Ports.
- Collaboration structures among a set of computational objects are modeled by Compositions with associated Choreographies.

It makes use of the Entities Model for the definition of entity components, where entity components correspond to entities in the information specification.

It makes use of the Events Model for the definition of event driven computational structures.

2.5.3 Levels of ProcessComponent in a Computational Specification

An EDOC computational specification can specify ProcessComponents at a number of different levels. These levels correspond to four general categories of ProcessComponent:

- E-Business Components
- Application Components
- Distributed Components
- Program Components

2.5.3.1 E-Business Components

E-Business Components are used as the integration point between enterprises, enterprises and customers or somewhat independent parts of a large enterprise (such as an acquired division). Interfaces to E-Business Components will frequently be directly accessible on the Internet as part of a web portal.

The E-Business Component has the potential to spawn new forms of business and new ways for business to work together.

E-Business Components integrate business entities that may share no common computing management or infrastructure. Interactions between E-Business components must be very loosely coupled and are always asynchronous. No assumptions of shared resources may be made between the parties, and the internals of the E-Business components will frequently be changed without informing other parties.

2.5.3.2 Application Components

Application Components represent new and legacy applications within an enterprise. Application Components are used to integrate applications (EAI) and create new applications, frequently to facilitate E-Business Components.

Application Components represent large-grain functional units. Each Application Component may be implemented in different technologies for different parts of the enterprise. Integrating Application Components facilitates enterprise-wide business processes and efficiencies.

Individual Application Components may be individually managed, but the integration falls under common management that may impose standards for interoperability and security.

Application Components use a wide variety of integration techniques including messaging, events, Internet exchanges and object or procedural RPC. Application Components are frequently wrapped legacy systems.

2.5.3.3 Distributed Components

Distributed Components are functional parts of distributed applications. These components are generally integrated within a common middleware infrastructure such as EJB, CORBA Components or DCOM. Distributed components have well defined interfaces and share common services and resources within an application.

Distributed Components provide for world-wide applications that can use a variety of technologies. Most distributed component interactions are synchronous.

2.5.3.4 Program Components

Program Components act within a single process to facilitate a program or larger grain component. Program Components may be technical in nature – such as a query component, or business focused – such as a “customer” component. These components will integrate under a common technology – such as J2EE.

Program Components provide the capability for drag-and-drop assembly of applications from fine-grain parts.

Note that some Program Components will provide access to the "outside world", such as CORBA or XML thus making a set of Program Components into a larger grain component.

The destination between Program Components and all others is quite important as these are the only components that do not use some kind of distributed technology – they are only used and visible within the context of "a program."

2.5.3.5 Relationships between ProcessComponent Levels

Relationships between ProcessComponent levels

Figure 2-3 shows how configurations of ProcessComponents at one level may *use and be composed of* ProcessComponents at lower levels. It also shows that at any level ProcessComponents may be primitive, that is – directly implemented without being a Composition. ProcessComponents may re-use and compose ProcessComponents at lower levels or the same level.

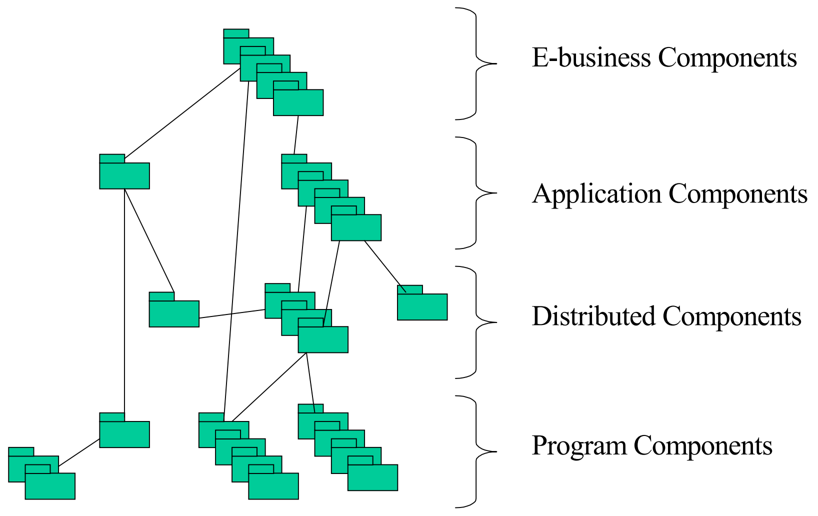


Figure 2-3 ProcessComponent Composition at multiple levels

There is no requirement or expectation that an EDOC computational specification must use all of these levels. For example, an E-Business Component could be directly composed of Program Components or it could use every levels.

2.6 Information Specification

2.6.1 Concepts

The information specification defines the semantics of information and information processing involved in the parts of the business processes carried out by the EDOC system (or by components that comprise that system). The information specification concepts are taken from the RM-ODP Part 3 Clause 6 (see Appendix A [3]).

The information specification is expressed in terms of

- a configuration of information objects (static schema),
- the behavior of those information objects (dynamic schema), and
- the constraints that apply to either of the above (invariant schema).

The information objects identified correspond to enterprise objects in the enterprise specification for which information is held and processed by the system.

The structure of the information objects and the relationships between them are defined in terms of static (structural) configurations of information objects. This includes the structure of individual information objects and the structure comprising a set of related information objects.

The behavior of the interrelated information objects is defined in terms of state changes that can occur and relate to the effects of the process steps in the enterprise specification.

The constraints relate to the business rules that apply to the process steps in the enterprise specification and define predicates on the information objects that must always be true.

2.6.2 EDOC Information Specifications

An EDOC information specification makes use of the Entities model and the Relationships model for the basic definition of the information structure, where:

- information objects are modeled as Entities and Relationships;
- constraints are defined in terms of enumerated states, relationship properties, and invariants from UML.

It makes use of the Choreography from the CCA for the definition of behavior of Entities in terms of changes of EntityState.

It makes use of the Relationships model for rigorous specification of relationships.

2.7 Engineering Specification

2.7.1 Concepts

The engineering specification defines the distribution transparency requirements and the services required to provide these transparencies in support of the processing specified by the computational specification. In addition, the engineering specification describes the means by which distribution is provided. The engineering specification concepts are taken from the RM-ODP Part 3 Clause 8 (see Appendix A [3]).

The engineering specification is derived from the computational specification by applying a technology mapping. The technology mapping incorporates standard interface and naming protocols to define consistent interface types and specifications.

The engineering specification will also incorporate additional design decisions. One of the key aspects of the engineering specification is the strategy for distributed computing, governing such issues as:

- which objects are network accessible and which are not: objects that are not network accessible must be co-located with objects with which they have relationships or from which they receive messages;
- the scope of transactions and the use of asynchronous messaging;
- which elements are persistent and how they are mapped to a persistent data store.

The engineering specification provides the basis for code generation. Currently, the ECA elements along with current UML design facilities can provide specifications for code to implement the objects, their interfaces, code to assure model integrity and methods to support certain services and protocols. Humans will still be required to program the business logic of methods and processes.

2.7.2 EDOC Engineering Specifications

These are defined by mapping from the computational specification to a technology abstraction model such as FCM. Examples of such mappings are given in Section II.

2.8 Technology Specification

The technology specification is concerned with the choice and deployment of software and hardware products for implementing the system and with the associated mappings from technology abstraction models such as FCM to the corresponding technologies (e.g. *J2EE with EJB,* Flow Composition Model (FCM), *CORBA 3 with CCM and MS DNA/.Net with DCOM)*.

2.9 Specification Integrity - Interviewpoint Correspondences

This section identifies relationships that are required to exist between viewpoint specifications and are expressed through relationships between elements in different viewpoint specifications.

2.9.1 Computational-Enterprise Interrelationships

A Process in the computational specification is related one or more sets of Activities in one or more Processes in the enterprise specification, where performance of those Activities is the responsibility of the EDOC system. It may also be related to Business Rules that apply to those Activities.

An Entity in the computational specification is related to a Entity referenced (as an artifact) in at least one Activity in a Process in the enterprise specification, where the Activity is the responsibility of the EDOC system.

A BusinessNotification in the computational specification is related to a BusinessNotification associated with an Activity in a Process in the enterprise specification, where the Activity is the responsibility of the EDOC system.

A Rule in the computational specification is related to a Rule that applies to Activities in one or more Processes in the enterprise specification, where the Activities are the responsibility of the EDOC system.

2.9.2 Computational-Information Interrelationships

A Entity in the computational specification is related to an entity or a configuration of Entities in a static schema in the information specification.

A Process in the computational specification is related to a Choreography in the information description and can be related also to an invariant schema.

A BusinessNotification in the computational specification is related to a Choreography in the information description.

A Rule in the computational specification is related to an invariant schema in the information specification.

2.9.3 Computational-Engineering Interrelationships

These depend upon the specific technology mappings that are applied.

2.9.4 Engineering-Technology Interrelationships

These depend upon the specific technology mappings that are applied.

The Enterprise Collaboration Architecture Model 3

Contents

This chapter includes the following topics.

Section I - ECA Design Rationale

This chapter describes the Enterprise Collaboration Architecture (ECA) – a model-driven architecture approach for specifying Enterprise Distributed Object Computing systems.

3.1 Key Design Features

Five key design features of the ECA address the EDOC vision:

- Recursive component composition;
- Support for event-driven systems;
- Process specification;
- Integration of process and information models;
- Technology independence, allowing implementation of a design using different technologies.

3.1.1 Recursive Component Composition

Business processes are by their very nature collaborations – a set of people, departments, divisions or companies, working together to achieve some purpose or set of purposes.

Such a collaboration can be viewed as a "composition" with the people, departments etc. as "components" of that composition having "roles" that represent how each component is to behave within the composition (note that the same component may have different roles in the same or different compositions, just as a person, department etc. may have many roles with respect to many processes).

This dynamic of component and composition is fundamental, the concept of component only makes sense with respect to some specific kind of composition and the concept of composition only makes sense when there can be components to compose it.

When a high-level business process is considered, such as buying and selling, there are roles within this buy-sell process for the buyer and seller. In some cases there may be other roles, such as banks, freight forwarders and brokers. Each of these is defined as a component within the high-level process (e.g., it is a component of the "buy-sell" process) playing some role.

Besides identifying the roles it is necessary to identify how each of the components must interact with the other components for the process to unfold. Thus, for each kind of interaction that exists between roles there is a protocol for that interaction defined by the information that flows and the timing of that flow, for example the interaction of the seller with a freight forwarder is completely different from the interaction with the buyer. This leads to the next important concept – that of interactions. Interactions are well defined protocols between roles within some composition. Each interaction point on a component is called a "port," which is the point of interaction of roles.

Finally, reflecting what is seen in the world, it is necessary to allow "drill down" from one level of granularity to another. When you place an order on the web you see a single face (the web portal) playing a single role (the seller). This simplified view represents the seller's role in the buy-sell process (you represent the other role). Inside of the seller, when it is opened up, you see order processing, credit, warehousing, shipping – all of the roles it takes to get you your order. This more fine-grain process represents the way a particular component has been configured to play the role of the seller, another seller may involve other choices.

Adding this concept of drill-down takes us from "flat" component composition to recursive component composition – the ability to define components as compositions of finer grain components.

Thus, components are defined in terms of sub-components playing roles and interacting through ports. At the highest level, processes are self-contained, the entire community of roles is identified. When you "open up" one of the components you may find a "primitive" component, one defined in terms of pre-established constructs such as may be found in Java or the UML Action language. The other thing you may find is another composition. What looks like an atomic component at one level may reveal a complex lattice of sub-components when "opened up."

A recursive component architecture can be used "top down", by defining new processes in terms of higher level compositions. It can also be used "bottom up" by assembling existing components into new compositions – making new components. As new basic capabilities are required they can either by defined from existing components or new primitive components can be supplied, so there is no "brick wall" when some fundamental capability you need was not anticipated.

In such a recursive component architecture there is a clear separation between the "inside" of a component and its "outside." The outside of a component exposes a set of named ports, each with a defined interaction that connects it with a compatible port in another component. These ports specify what information flows between the compatible components and under what conditions the information flows. The outside of a component is not concerned with the internal composition or process of the component.

One other aspect of component technology is that of configurability. Components may be very general in nature, which promotes reuse. These very general components must be configured when used in a specific role. This may be seen in the property panels of bean-boxes or COM components. The ability to configure a component is essential to making it general and reusable. Configuration points for process components are "properties" and "Contextual Bindings".

To summarize the points:

- The concepts of component and composition are fundamentally tied.
- Components may be primitive or compositions of sub-components.
- Each component can play roles within other compositions.
- Components interact with each other, within composite processes, through ports.
- Component composition is recursive, allowing decomposition and assembly.

The advantages of this approach are:

- A single simple paradigm describes large grain and fine grain process components.
- Components are reusable across many compositions.
- New components may be defined as collaborations of existing components.
- New fundamental capabilities may defined as primitive components.
- The collaborative and recursive nature of processes may be directly represented.

3.1.2 Process Specification

The Business Process model specializes the CCA, and describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model system behavior in the context of the business it supports.

The model provides modeling concepts that allow the description of business processes in terms of a composition of business activities, selection criteria for the entities that carry out these activities, and their communication and coordination. In particular, the Business Process model provides the ability to express:

- Complex dependencies between individual business tasks (i.e., logical units of work) constituting a business process, as well as rich concurrency semantics.
- Representation of several business tasks at one level of abstraction as a single business task at a higher level of abstraction and precisely defining relationships between such tasks, covering activation and termination semantics for these tasks.
- Representation of iteration in business tasks.
- Various time expressions, such as duration of a task and support for expression of deadlines.

- Support for the detection of unexpected occurrences while performing business tasks that need to be acted upon, (i.e., exceptional situations).
- Associations between the specifications of business tasks and business roles that perform these tasks and also those roles that are needed for task execution.
- Initiation of specific tasks in response to the occurrence of business events.
- The exposure of actions that take place during a business process as business events.

The modeling of processes in the ECA model addresses an RFP requirement, but, more importantly, processes are important elements in the representation of interactions between components, systems and enterprises. Processes are the mechanisms of collaborations. Processes define the roles of the participants and artifacts involved in collaborations. Processes also define the manner in which events can drive the operation of the enterprise. Consequently, it is essential that the ECA model include a representation of processes that enables a modeler to define a framework for the operation of an enterprise.

The modeling of processes in the ECA model reflects the OMG Workflow Management Facility model. A process contains activities, which perform the actions of the process. The activities may invoke other processes, and they may employ resources. The Workflow Management Facility resource interface represents the participation of that resource, i.e., a role in the ECA context. The resource/role captures the state and supports the interaction between the activity and a potentially wide variety of resources.

The ECA model goes slightly beyond the Workflow Management Facility specification. First, it extends the resource concept by defining performers and artifacts (active and passive participants). Second, it adds the ability to attach pre and post conditions to activities. These are concepts that are consistent with workflow management concepts and provide basic flow control mechanisms. These were not addressed in the Workflow Management Facility specification because it focused primarily on interoperability between workflow management systems.

The ECA model does not attempt to define a representation of the action semantics of processes, nor does it define the relationship of processes to organizations or applications. These are left to other RFPs to be addressed by specialists in these areas.

3.1.3 Specification of Event Driven Systems

Event driven computing is becoming the preferred distributed computing paradigm in many enterprises and in many collaborations between enterprises. Event driven computing combines two kinds of loosely coupled architectures.

The first one is loosely coupled, distributed components that communicate with each other through asynchronous messaging.

The other one is loosely coupled business process execution. Here enterprises collaborate under an overall long term contract, but do not execute their day to day interaction in traditional workflow, or request/response style interaction.

In event driven computing the most important aspect of a process is the events that happen during its execution, and the most important part of the component-to-component communication is the notification of such events from the party that made them happen to all the parties that need to react to them.

In ECA we support both the definition of loosely coupled business processes, as well as the loosely coupled communication between distributed components.

Neither the world, nor the computing world, however, apply only one paradigm to their problem space. Businesses use a combination of loosely coupled and tightly coupled processes, and computing solutions deploy a combination of loosely coupled and tightly coupled styles of communication and interaction between distributed components.

An ECA process can be defined as event driven for some of its steps and workflow or request/response driven for others. ECA distributed components can be configured to communicate with each other in a mixture of event-driven publish-and-subscribe, asynchronous peer-to-peer, and client-server remote invocation styles.

The essential elements of the purely event driven approach are:

- Business Process objects are configured with a set of Business Rule parameters that determine what Business Events trigger actions, and what the action should be.
- Business Process objects operate on Business Entity objects that represent people, products, and other business resources and artifacts.
- When actions are performed on Business Entity objects, Business Events happen.
- All Business Entity objects are capable of notifying the world of events that happen to them.
- All Business Process objects are capable of subscribing to such events and interpreting them throughout their set of business rules.

3.1.4 Integration of Process and Information Models

IT systems are specified with entity and process models, where entity models describe the things (entities, attributes, relationships, invariants) in the IT system and process models specify the processes, sub-processes, activities, resources, roles, and rules of IT system behavior.

Information modeling tools, such as those based on the UML metamodel, are used to specify entity models. Process definition tools, such as those provided by BPR and workflow vendors, are used to specify process models. As these entity model and process model tools are based on different metamodels, the integration of their models into the IT system specification is a problem.

IT system designers and developers typically work round the problem by looking at one model, then the other, and then do their own composition for that moment (perhaps influenced by memories of other compositions). One result of this is that the normative entity and process models when composed, by each individual at multiple moments in time, become non-normative individual interpretations of the IT system specification.

Also of concern is the impact of model changes to the composition – evolution of process and entity models is reasonably certain, especially during IT system development projects.

With this metamodel UML, workflow, and BPR vendors can provide new tools that combine entity-orientated and process-orientated modeling techniques to produce integrated IT system models.

3.1.5 Rigorous Relationship Specification

3.1.6 Mappings to Technology - Platform Independence

Viewpoint abstractions in the context of model-based development provide mechanisms for specifying platform independent models of business applications.

Such platform independence is an important element in systems that can adapt to change and, hence, is a fundamental element of the EDOC vision (). The rate of change of today's enterprises and their requirements generates demands for flexible and dynamic systems that are capable of coping with the ever changing business requirements and with changes in software and hardware technologies.

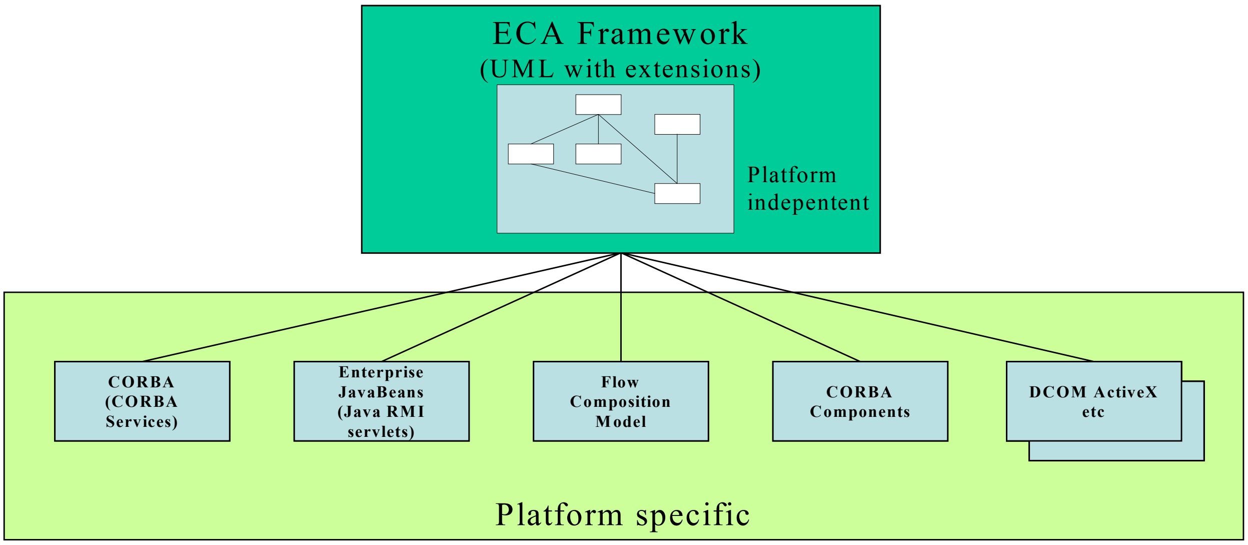


Figure 3-1 EDOC framework vision

3.2 ECA Elements

The Enterprise Collaboration Architecture (ECA) comprises a set of five models. Each model consists of a set of model elements that represent concepts needed to model specific aspects of EDOC systems and address specific aspects of the key design features. The concepts are described in terms of Models. Each model is also expressed as a UML Profile in a separate document - "UML Profile for ECA."

These models are defined in the remainder of this chapter:

- Component Collaboration Architecture (CCA) - details how to model at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system – See Section II.
- Entities model - describes a metamodel that may be used to model entity objects that are representations of concepts in the application problem domain and define them as composable components – See Section III.
- Events model - describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model event driven systems – See Section IV.
- Business Process model - specializes the CCA and describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model system behavior in the context of the business it supports – See Section V.

The ECA models are technology independent and are used together to define platform independent models of EDOC systems in conformance with the MDA. In particular, they enable the modeling of the concepts that until now have had to be specified programmatically in terms of the use of services such as events/ notification, support for relationships and persistence.

Section II - The Component Collaboration Architecture

The Component Collaboration Architecture (CCA) details how to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system.

3.3 Rationale

3.3.1 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.

3.3.1.1 Recursive decomposition and assembly

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

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

Virtually every project involves both top-down decomposition in specification and "bottom up" assembly of existing parts. Bringing together top-down specification and bottom-up assembly is the challenge of information system engineering.

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

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

3.3.1.2 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.

3.3.1.3 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.

3.3.1.4 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.

3.3.1.5 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.

3.3.1.6 Enabling a business component Marketplace

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

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

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

3.3.1.7 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 solve my business problems. Simplistic infrastructure and tools that make it hard to solve real problems are not viable.

3.3.2 Concepts

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

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

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

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

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

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

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

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

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

3.3.2.2 ProcessComponent Libraries

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

3.3.2.3 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 ProcessComponents, and a *technology profile*, it will be executable.

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

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

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

3.3.2.4 Specification Vs. Methodology

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

3.3.2.5 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 (the UML Profile for ECA) when such is preferred or CCA specific tooling is not available. The CCA notation can be used to achieve greater simplicity and economy of expression.

3.3.3 Conceptual Framework

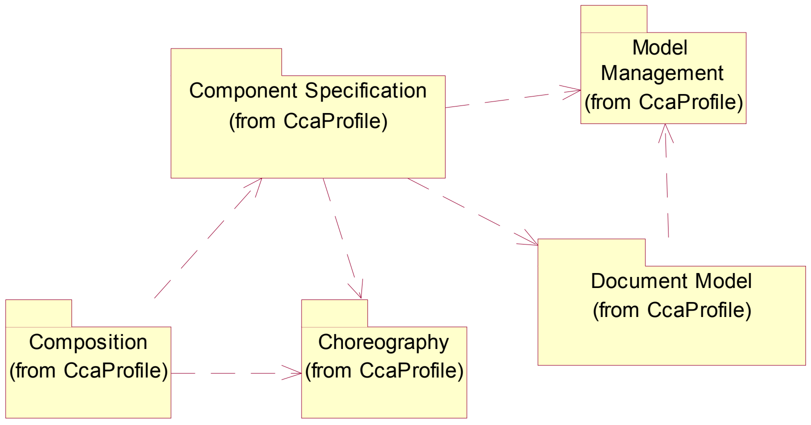

Model Management (from CcaProfile)
Component Specification (from CcaProfile)
Document Model (from CcaProfile)
Composition (from CcaProfile)
Choreography (from CcaProfile)

Figure 3-2 Structure and dependencies of the CCA Metamodel

3.3.3.1 ProcessComponent Specification

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

3.3.3.2 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.

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

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

3.3.3.3 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 a **ProcessComponent** and various ways to “bind” the correct implementation when a component is used.

The two basic kinds of concrete components are:

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

3.3.3.4 Composition

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

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

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

3.3.3.5 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 ProcessComponents (see the Entities model, Section III, below).

3.3.3.6 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.

3.4 CCA Metamodel

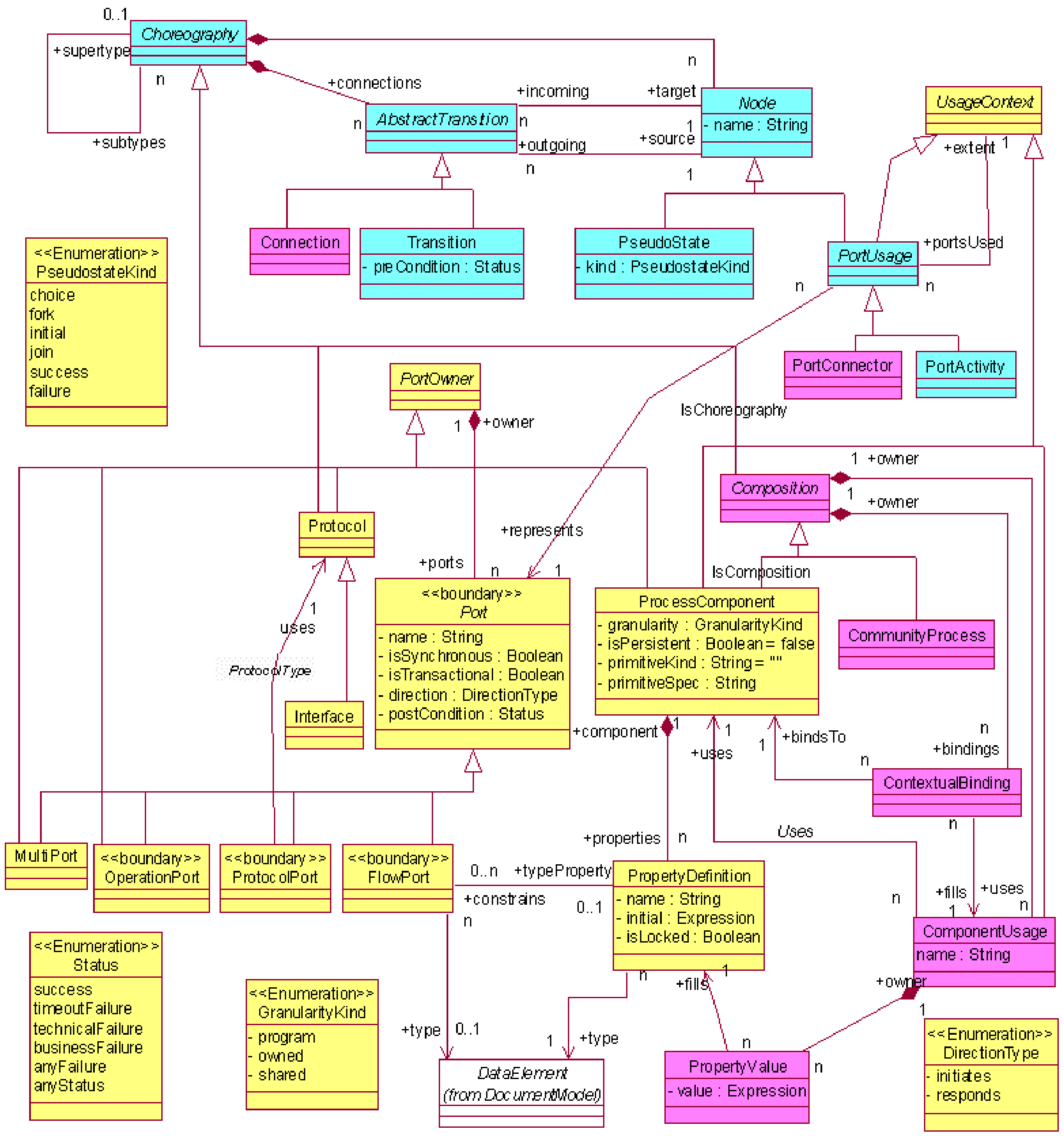


Figure 3-3 CCA Major Elements

Figure 3-3 is a combined model of the major elements of the CCA component specification defined below.

3.4.1 Structural Specification

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

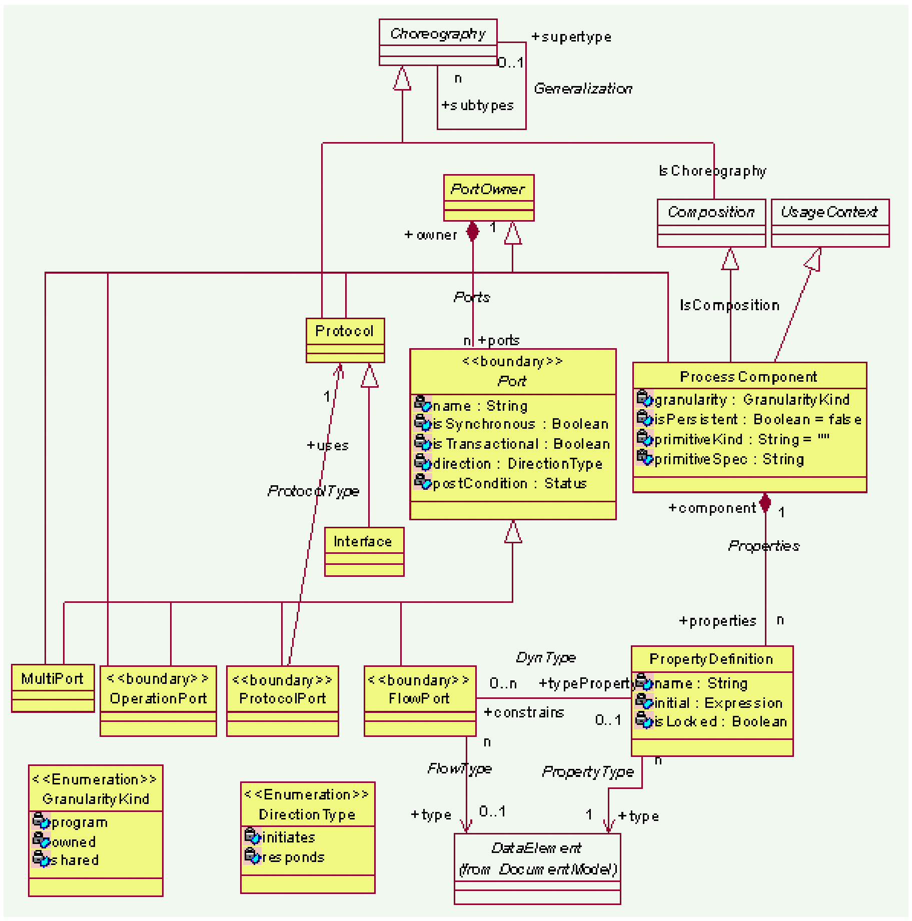


Figure 3-4 Structural Specification Metamodel

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

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

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

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

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

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

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

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

3.4.1.1 ProcessComponent

Semantics

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

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

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

Fully Scoped name

ECA::CCA::ProcessComponent

Owned by

Package

Extends

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

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

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

Properties

Granularity

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

- **Program** – the component is local to a program instance (default).
- **Owned** – the component is visible outside of the scope of a particular program but dedicated to a particular task or session that 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.

isPersistent

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

primitiveKind

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

primitiveSpec

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

Related elements

Ports (via "PortOwner")

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

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

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

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

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.

Constraints

A process component may only inherit from another process component.

3.4.1.2 Port

Semantics

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

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

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

Business Example: Flight reservation Port

Fully Scoped name

ECA::CCA::Port

Owned by

ProcessComponent or Protocol via PortOwner

Extends

None

Properties

isTransactional

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

isSynchronous

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

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.

Direction

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

- **Initiates** – this port will initiate the conversation by sending the first message..
- **Responds** – this port will respond to the initial message and (potentially) continue the conversation.

PostCondition

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

Related elements

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

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

Constraints

None

3.4.1.3 FlowPort

Semantics

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

Fully Scoped name

ECA::CCA::FlowPort

Owned by

PortOwner

Extends

Port

Properties

None

Related elements

type

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

TypeProperty

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

Constraints

None

3.4.1.4 ProtocolPort

Semantics

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

Fully Scoped name

ECA::CCA::ProtocolPort

Owned by

PortOwner

Extends

Port

Properties

None

Related elements

uses

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

Constraints

None

3.4.1.5 OperationPort

Semantics

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

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

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

Fully Scoped name

ECA::CCA::OperationPort

Owned by

PortOwner (Protocol or ProcessComponent)

Extends

Port and PortOwner

Properties

None

Related elements

Ports (Via PortOwner)

The flow ports representing the call and returns.

Constraints

As a PortOwner, the OperationPort:

- May only contain FlowPorts.
- Must contain exactly one flow port with direction set to “responds.”
- Must contain exactly one flow port with direction set to “initiates” (the call).

3.4.1.6 MultiPort

Semantics

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

Fully Scoped name

ECA::CCA::MultiPort

Owned by

PortOwner

Extends

Port & PortOwner

Properties

None

Related elements

Ports (Via PortOwner)

The flow ports owned by the MultiPort.

Constraints

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

3.4.1.7 Protocol

Semantics

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

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

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

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

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

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

Fully Scoped name

ECA::CCA::Protocol

Owned by

Package

Extends

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

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

Properties

None

Related elements

Ports (Via PortOwner)

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

Constraints

None

3.4.1.8 Interface

Semantics

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

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

Fully Scoped name

ECA::CCA::Interface

Owned by

Package

Extends

Protocol

Properties

None

Related elements

Ports (Via Protocol & PortOwner)

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

Initiator (Via Protocol)

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

Responder (Via Protocol)

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

Constraints

The Ports related by the "Ports" association must

- be of type OperationPort or FlowPort
- have direction == "responds."

3.4.1.9 PropertyDefinition

Semantics

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

Fully Scoped name

ECA::CCA::PropertyDefinition

Owned by

ProcessComponent

Extends

None

Properties

name

Name of the property being modeled.

initial

An expression indicating the initial & default value.

isLocked

The property may not be changed.

Related elements

component

The owning component.

type

The type of the property.

Constraints

If the “constrains” relation contains any links, the PropertyValue must contain the fully qualified name of a DataElement.

3.4.1.10 PortOwner

Semantics

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

Fully Scoped name

ECA::CCA::PortOwner

Owned by

None

Extends

None

Related elements

ports

The owned ports

Constraints

None

3.4.2 Choreography

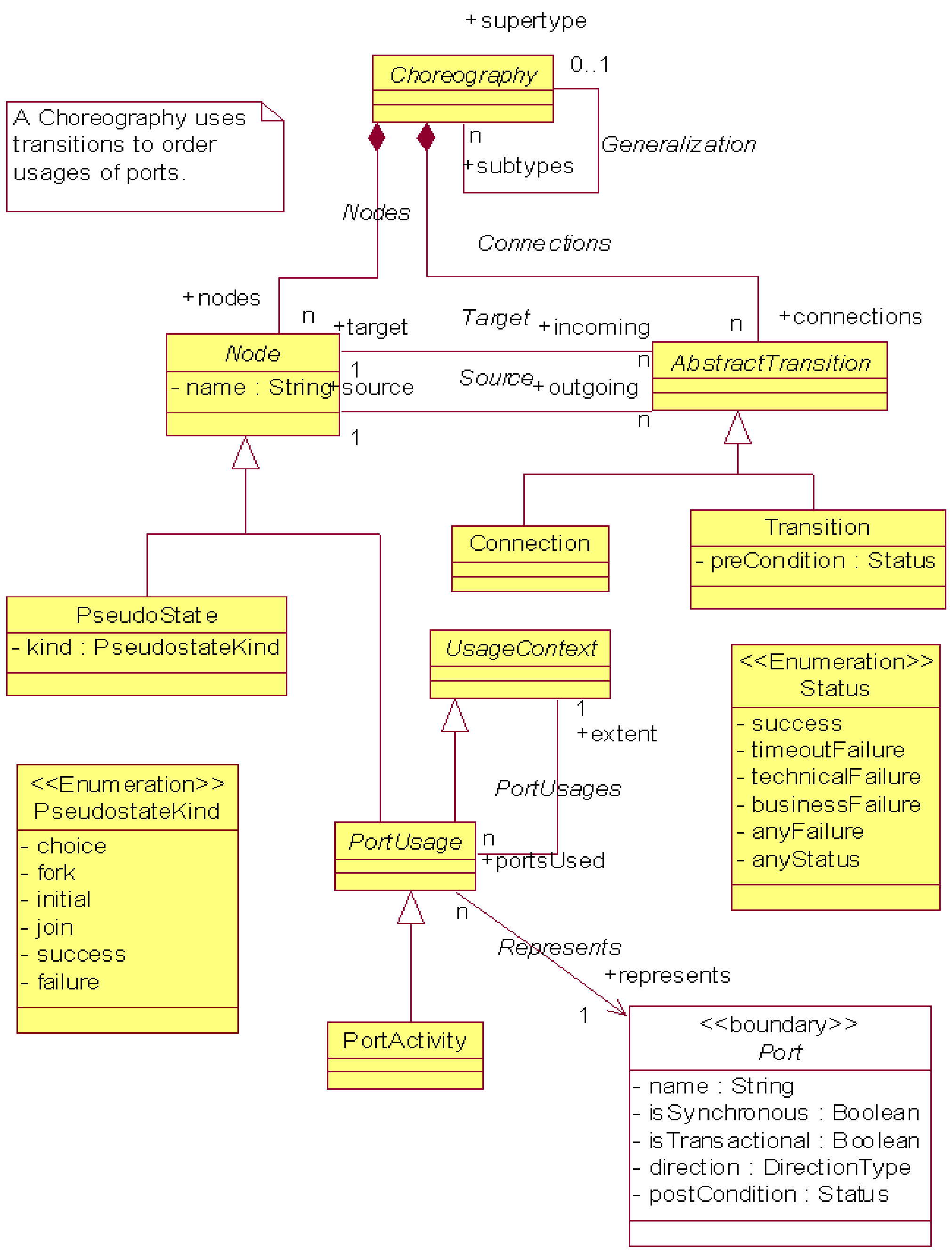


Figure 3-5 Choreography Metamodel

A Choreography specifies how messages will flow between PortUsages. The choreography may be externally oriented, specifying the contract a component will have with other components or, it may be internally oriented, specifying the flow of messages within a composition. External chirographies are shown as an activity graph while internal choreography is shown as part of a collaboration. An external choreography may be defined for a protocol or a ProcessComponent.

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

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

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

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

3.4.2.1 Choreography

Semantics

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

Fully Scoped name

ECA::CCA::Choreography

Owned by

None

Extends

None

Properties

None

Related elements

Nodes

The states and port usages to be choreographed.

PortActivity ::SubmachineState

AbstractTransitions

The connections and transitions between nodes.

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

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

A component may be substituted by a subtype.

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

3.4.2.2 Node

Semantics

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

Fully Scoped name

ECA::CCA::Node

Owned by

Choreography

Extends

None

Properties

name

Related elements

Choreography

The owning protocol or ProcessComponent.

Incoming

Transitions that cause this node to become active.

outgoing

Nodes that may become active after this node completes.

Constraints

None

3.4.2.3 AbstractTransition

Semantics

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

Fully Scoped name

ECA::CCA::AbstractTransition

Owned by

Choreography

Extends

None

Properties

None

Related elements

Choreography

The owning choreography.

Source

The node which is transferring control and/or data.

Target

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

Constraints

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

3.4.2.4 Transition

Semantics

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

Fully Scoped name

ECA::CCA::Transition

Owned by

Choreography

Extends

AbstractTransition

Properties

preCondition

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

Related elements

Choreography (Via AbstractTransition)

The owning choreography.

Source

The node that is transferring control and/or data.

Target

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

Constraints

A transition may not connect PortConnectors.

3.4.2.5 PortUsage

Semantics

The usage of a port as part of a choreography.

Fully Scoped name

ECA::CCA::PortUsage

Owned by

Choreography

Extends

Node & Usage Context

Properties

None

Related elements

extent

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

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

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

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

Represents

The port that the PortUsage uses.

Constraints

None

3.4.2.6 UsageContext

Semantics

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

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

- PortUsages – representing ports nested via protocols.

Fully Scoped name

ECA::CCA::UsageContext

Owned by

None

Extends

None

Properties

None

Related elements

PortsUsed

Provides context for port usage

Constraints

None

3.4.2.7 PortActivity

Semantics

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

Fully Scoped name

ECA::CCA::PortActivity

Owned by

Protocol or ProcessComponent via Choreography

Extends

PortUsage

Properties

None

Related elements

None

Constraints

Port Activities may only be connected using transitions.

3.4.2.8 PseudoState

Semantics

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

Fully Scoped name

ECA::CCA::PseudoState

Owned by

Choreography

Extends

Node

Properties

Kind ; PseudostateKind

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

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

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

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

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

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

Related elements

None

Constraints

PseudoStates may only be connected using transitions.

3.4.3 Composition

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

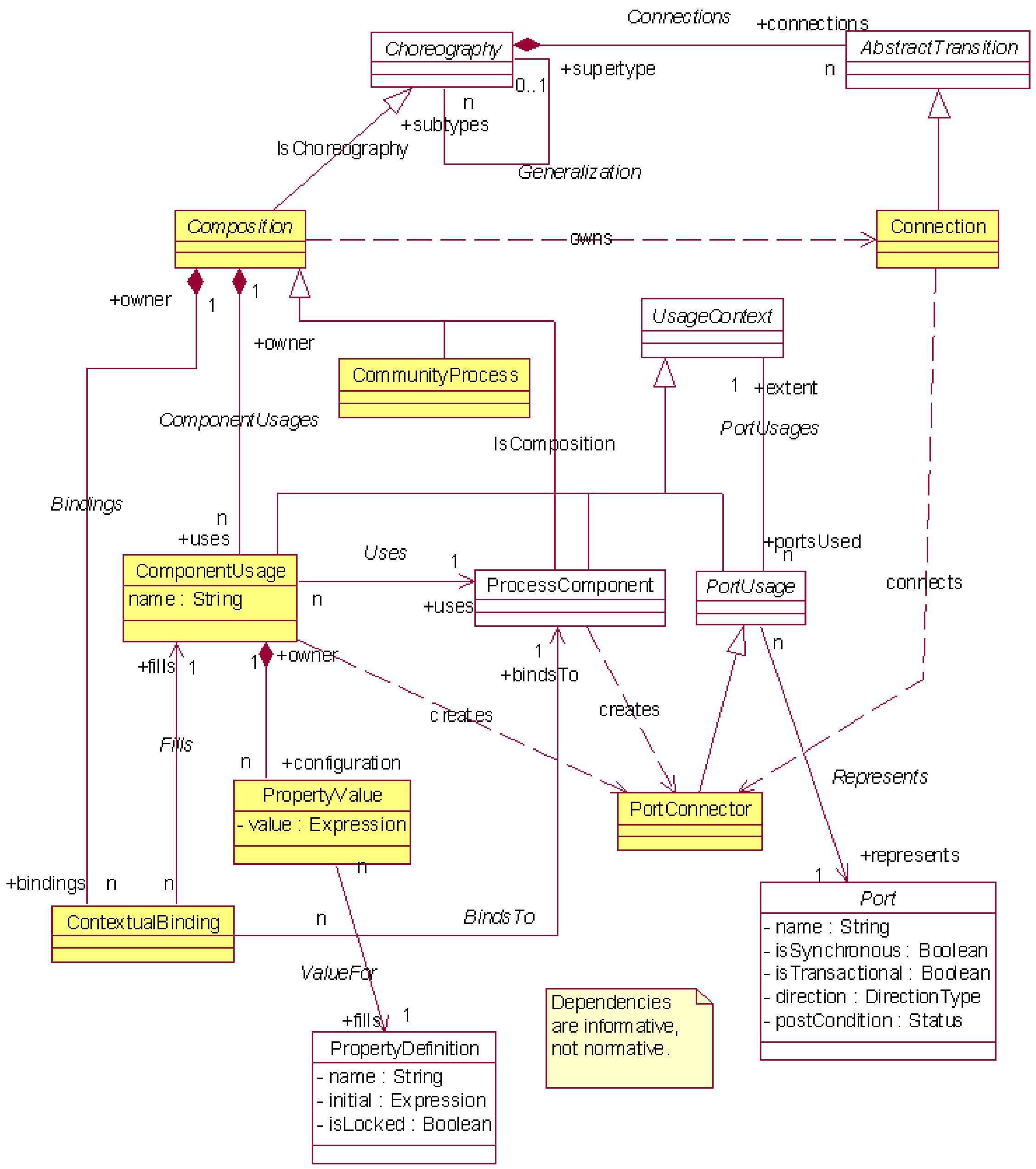


Figure 3-6 Composition metamodel

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

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

A composition also contains a set of “**Connections**.” A connection joins compatible ports on ProcessComponents together to define a flow of data. The other side will receive anything sent out of one side. So a Connection is a form of logical event registration (one-way registration for a flow port or Operation port, two-way registration for a ProtocolPort).

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

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

3.4.3.1 Composition

Semantics

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

Fully Scoped name

ECA::CCA::Composition

Owned by

None

Extends

Choreography

Properties

None

Related elements

bindings

ContextualBindings defined within the context of the composition.

uses

ComponentUsages defined within the context of the composition.

Connection (via choreography and AbstractTransition)

The flow of data and control between port connectors.

PortConnector (via Choreography and nodes)

The port instances to be connected by Connections.

Constraints

None

3.4.3.2 ComponentUsage

Semantics

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

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

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

Fully Scoped name

ECA::CCA::ComponentUsage

Owned by

Composition

Extends

UsageContext

Properties

Name

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

Related elements

owner

The owning composition

Uses

The type of ProcessComponent to use.

PortsUsed (Via UsageContext)

PortConnectors for each port on the used component.

Constraints

None

3.4.3.3 PortConnector

Semantics

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

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

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

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

Fully Scoped name

ECA::CCA::PortConnector

Owned by

Composition

Extends

PortUsage

Properties

None

Related elements

Represents (via PortUsage)

The port of which this is a port.

Contexts (via PortUsage)

The associated owner of the port.

Incoming and Outgoing Connections (Via PortUsage and Node)

The Connections.

Constraints

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

3.4.3.4 Connection

Semantics

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

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

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

Fully Scoped name

ECA::CCA::Connection

Owned by

Composition

Extends

AbstractTransition

Properties

None

Related elements

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

The PortConnectors between which the Connection is being defined.

Constraints

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

3.4.3.5 PropertyValue

Semantics

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

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

Fully Scoped name

ECA::CCA::PropertyValue

Owned by

ComponentUsage

Extends

None

Properties

value

An expression for the value of the property.

Related elements

Owner

The component usage being configured with a value.

Fills

The property being modified.

Constraints

"fills" must relate to a property definition of the ProcessComponent that the owner uses.

The type returned by the PropertyValue expression must be compatible with the type defined by the PropertyDefinition.

3.4.3.6 ContextualBinding

Semantics

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

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

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

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

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

Fully Scoped name

ECA::CCA::ContextualBinding

Owned by

Composition

Extends

None

Properties

None

Related elements

owner

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

fills

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

bindsTo

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

Constraints

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

3.4.3.7 CommunityProcess

Semantics

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

Components within a community process represent roles that components may play within the process. These roles are later bound to particular components realizing processes.

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

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

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

Fully Scoped name

ECA::CCA::CommunityProcess

Owned by

Package

Extends

Composition and Package

Properties

None

Related elements

None

Constraints

None

3.4.4 Document Model

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

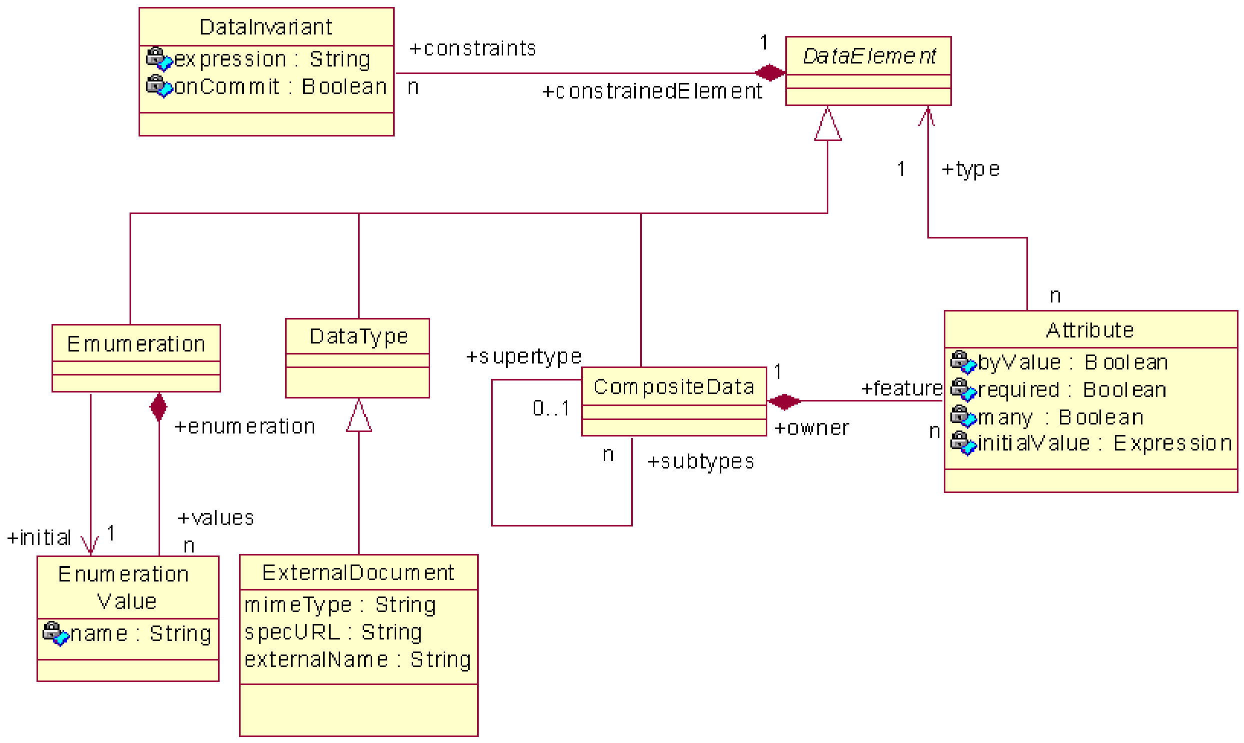


Figure 3-7 Document Metamodel

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

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

3.4.4.1 DataElement

Semantics

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

Fully Scoped name

ECA::DocumentModel::DataElement

Owned by

Package

Extends

PackageContent

Properties

None

Related elements

constraints

Constraints applied to the values of this data type.

Constraints

None

3.4.4.2 DataType

Semantics

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

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

Fully Scoped name

ECA::DocumentModel::DataType

Owned by

Package

Extends

DataElement

Properties

None

Related elements

None

Constraints

None

3.4.4.3 Enumeration

Semantics

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

Fully Scoped name

ECA::Documentmodel::Enumeration

Owned by

Package

Extends

DataElement

Properties

None

Related elements

Values

The set of values the enumeration may have.

Initial

The initial, or default, value of the enumeration.

Constraints

None

3.4.4.4 EnumerationValue

Semantics

A possible value of an enumeration.

Fully Scoped name

ECA::DOCUMENTMODEL::EnumerationValue

Owned by

Enumeration

Extends

None

Properties

name

Related elements

Enumeration

The owning enumeration.

Constraints

None

3.4.4.5 CompositeData

Semantics

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

Fully Scoped name

ECA::DocumentModel::CompositreData

Owned by

Package

Extend

DataElements

Properties

None

Related elements

Feature

The attributes that 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

3.4.4.6 Attribute

Semantics

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

Fully Scoped name

ECA::DOCUMENTMODEL::Attribute

Owned by

CompositeData

Extends

None

Properties

isByValue

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

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 attribute.

Related elements

type

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

owner

The composite of which this is an attribute.

Constraints

None

3.4.4.7 DataInvariant

Semantics

A constraint on the legal values of a data element.

Fully Scoped name

ECA::DOCUMENTMODEL::DataInvarient

Owned by

DataElement

Extends

None

Properties

Expression

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

isOnCommit (Default: False)

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

Related elements

ConstrainedElement

The data element that will be constrained.

3.4.4.8 ExternalDocument

Semantics

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

Fully Scoped name

ECA::DOCUMENTMODEL::ExternalDocument

Owned by

Package

Extends

DataElement

Properties

All properties are tagged values.

MimeType

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

SpecURL

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

ExternalName

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

Related elements

None

Constraints

None

3.4.5 Model Management

Model management defines how CCA models are structured and organized.

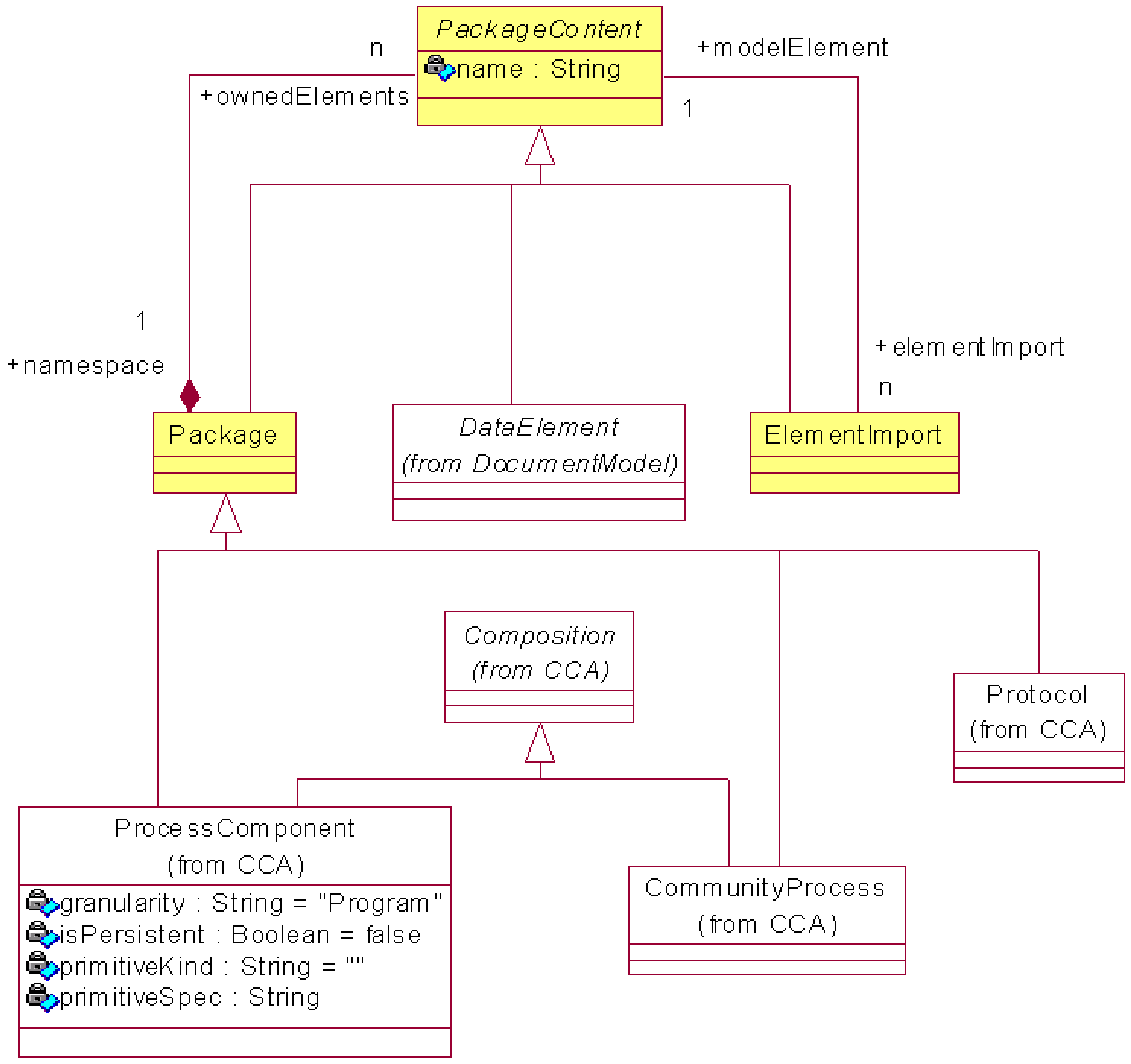


Figure 3-8 Model Management Metamodel

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

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

3.4.5.1 Package

Semantics

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

Fully Scoped name

ECA::ModelManagement::Package

Owned by

Package or model (global scope)

Extends

PackageContent

Properties

None

Related elements

OwnedElements

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

Constraints

None

3.4.5.2 PackageContent

Semantics

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

Fully Scoped name

ECA::ModelManagement::

Owned by

Package

Extends

None

Properties

name

Related elements

namespace

Constraints

3.4.5.3 ElementImport

Semantics

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

Fully Scoped name

ECA::ModelManagement::ElementImport

Owned by

Package

Extends

PackageContent

Properties

None

Related elements

ModelElement

The element to be imported.

Constraints

None

3.5 CCA Notation

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

3.5.1 CCA Specification Notation

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

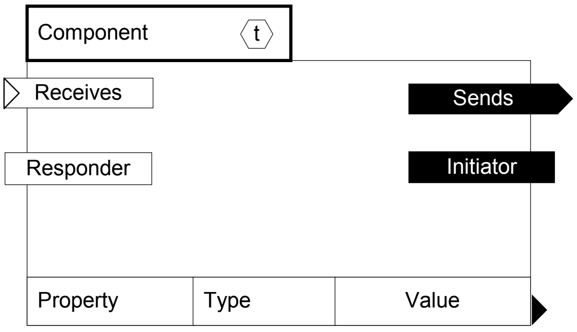


Figure 3-9 ProcessComponent specification notation

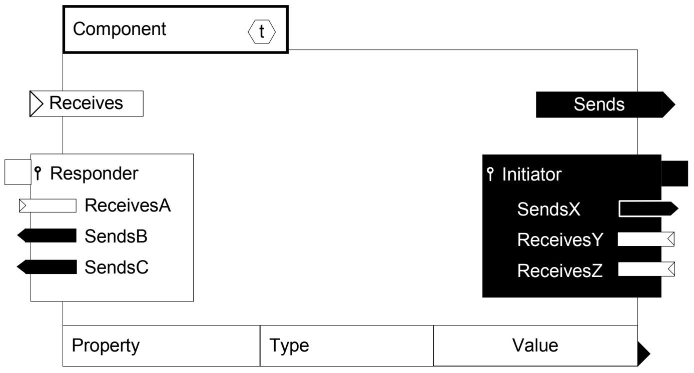


Figure 3-10 ProcessComponent specification notation (expanded ProtocolPorts)

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

- **Protocol ports** and **Operation ports** are boxes extending out of the component. Protocol ports representing an initiator have the colors of their background and text reversed. In the above, “Initiator” is a protocol port of an initiator and “Responder” is a protocol port that is not an initiator. ProtocolPorts may show nested, the Ports of the used Protocol.
- **Multiports** are shown as a shaded box grouping the set of ports it contains.
- **Property Definitions** are in a separate compartment listing the property name, type, and default value (if any). The name, type, and value are separated by lines. Each property is on a separate line.

3.5.2 Composite Component Notation

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

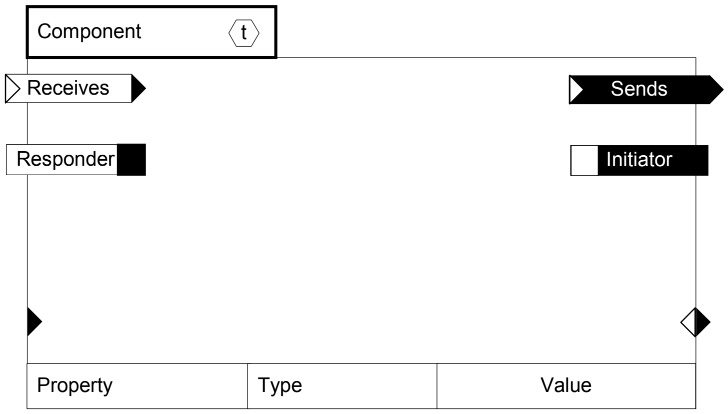


Figure 3-11 - Composite Component notation (without internal ComponentUsages)

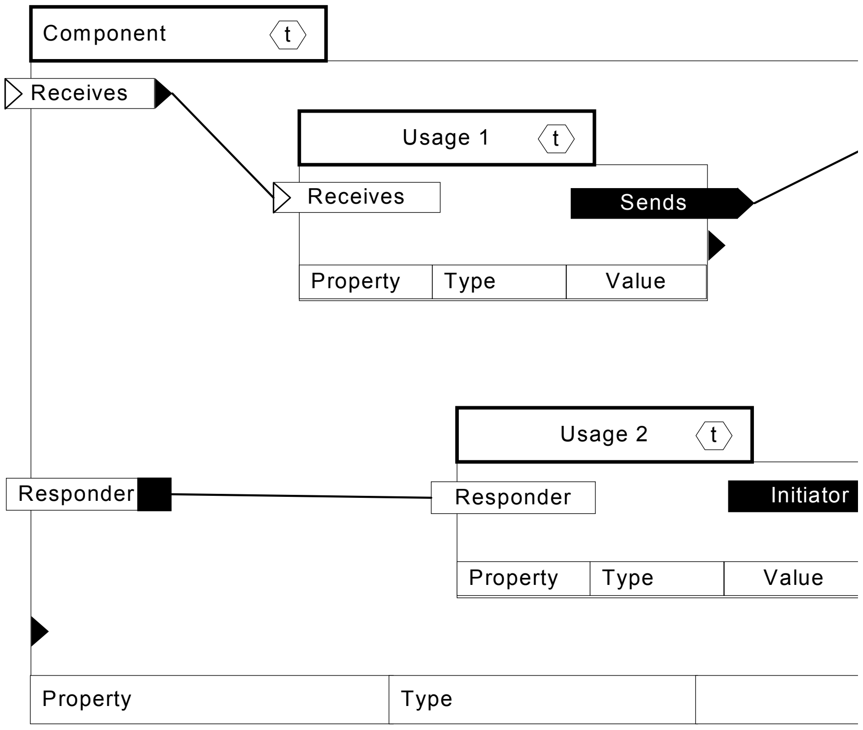


Figure 3-12 - Composite Component notation

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

3.5.3 Community Process Notation

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

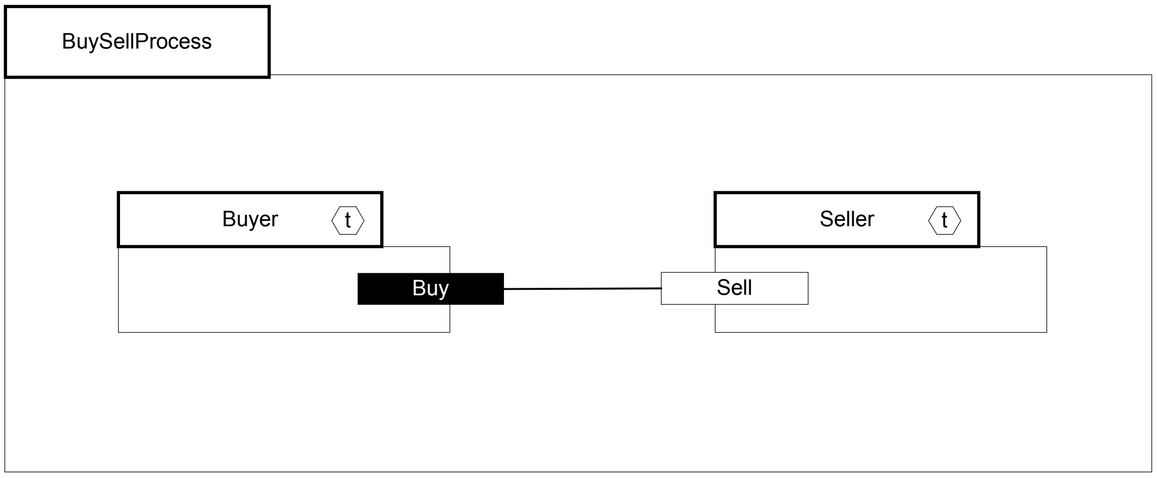


Figure 3-13 Community Process notation

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.

3.6 Diagramming CCA

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

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

3.6.1 Types of Diagram

The diagrams used to express CCA concepts are as follows.

3.6.1.1 Class Diagrams for the Document Model

These are used to express the document model.

3.6.1.2 *Class Diagrams for the Component Structure*

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

3.6.1.3 *Collaboration Diagrams for Composition*

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

3.6.1.4 *State or Activity Diagrams for Protocols & Process Components*

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

3.6.1.5 *CCA Notation for Process Component Structure & Composition*

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

3.6.2 *The Buy/Sell Example*

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

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

3.6.3 *Collaboration Diagram Shows Community Process*

At the highest level we show a collaboration diagram of the Buy/Sell community process. In the design tool we also created a package for this process to hold the relevant model elements. See Figure 3-20.

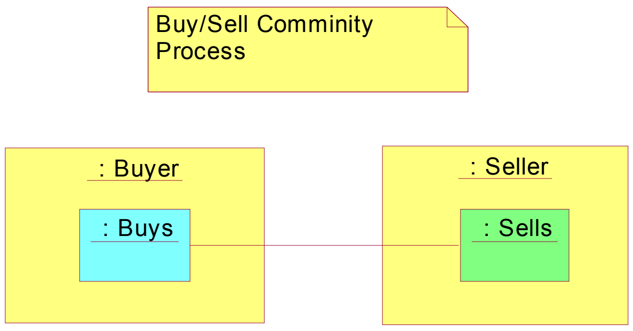


Figure 3-20 Top Level Collaboration Diagram

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

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

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

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

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

3.6.4 Class Diagram for Protocol Structure

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

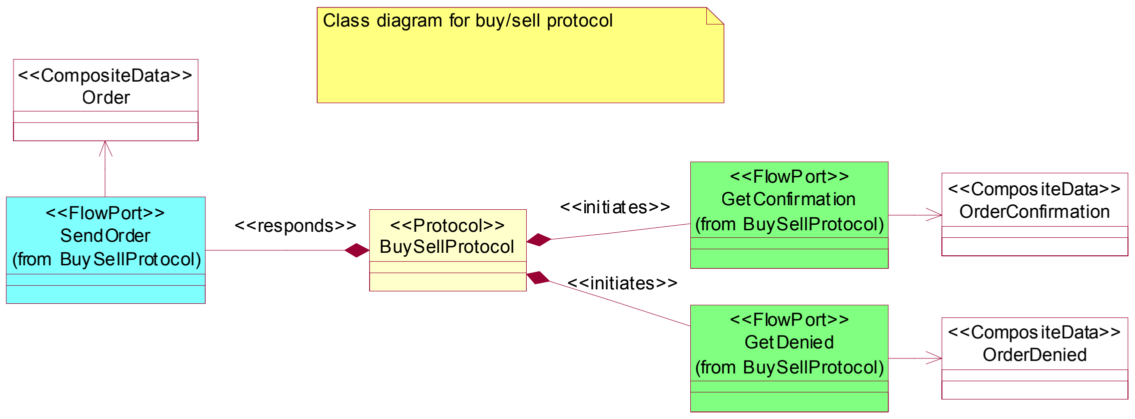


Figure 3-21 Class diagram for protocol structure

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

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

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

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

3.6.5 Activity Diagram (Choreography) for a Protocol

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

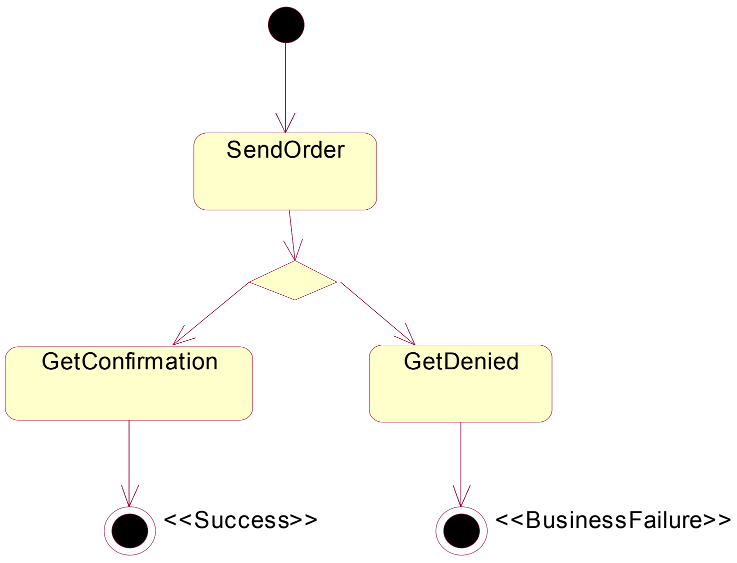


Figure 3-22 Protocol Activity Diagram

3.6.5.1 Choreography of a Protocol

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

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

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

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

3.6.6 Class Diagram for Component Structure

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

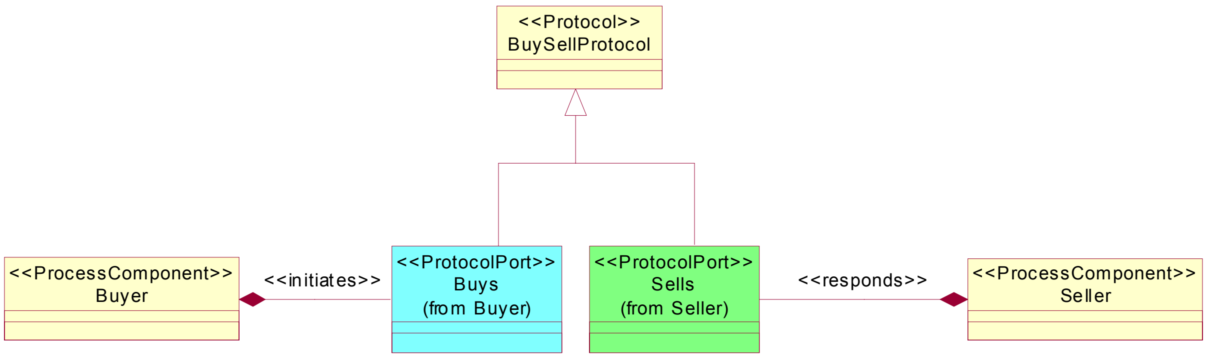


Figure 3-23 Class Diagram for Component Structure

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

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

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

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

3.6.7 Class Diagram for Interface

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

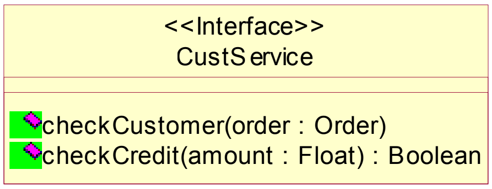

<<Interface>>
CustService
checkCustomer(order : Order)
checkCredit(amount : Float) : Boolean

Figure 3-24 Class Diagram for Interface

Since the semantics of such an interface are will understood, let’s just relate to the CCA elements, as shown below.

Table 3-1 Elements of an Interface

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

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

Interfaces may have the same tagged values as protocol, but interfaces don’t need “direction,” the direction is always “responds.”

3.6.7.1 *Using Interfaces*

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

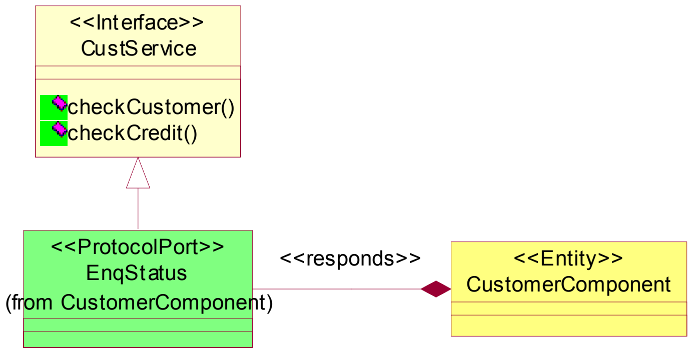


Figure 3-25 Using Interfaces

This diagram shows an “Entity” ProcessComponent (see entity model) called “CustomerComponent,” which exposes a ProtocolPort (EnqStatus) that implements this interface.

3.6.8 Class Diagram for Process Components with Multiple Ports

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

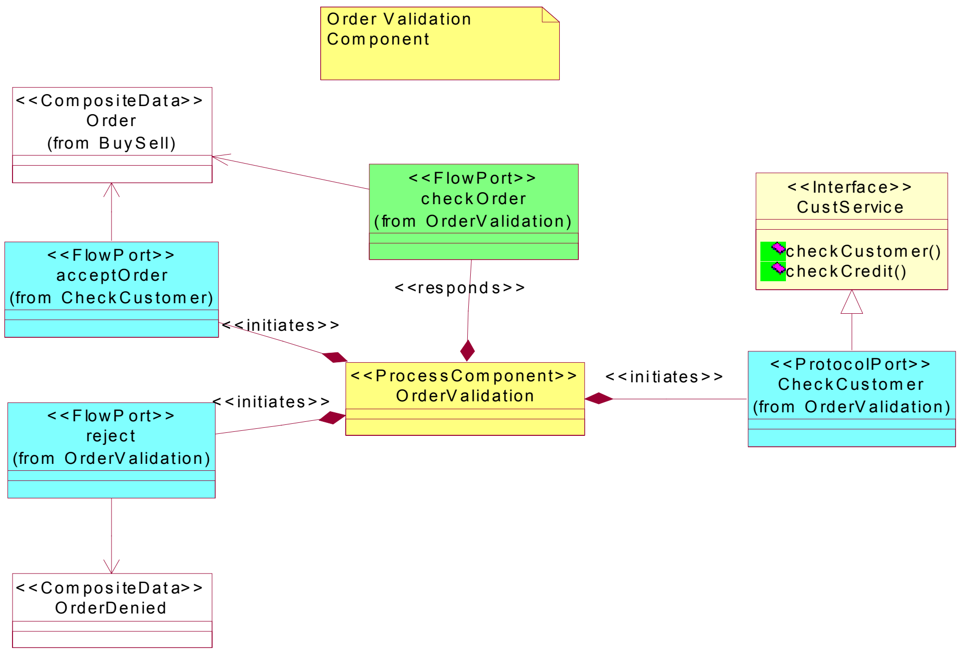

Order Validation Component
<<CompositeData>> Order (from BuySell)
<<FlowPort>> checkOrder (from OrderValidation)
<<Interface>> CustService
checkCustomer()
checkCredit()
<<FlowPort>> acceptOrder (from CheckCustomer)
<<responds>>
<<initiates>>
<<ProcessComponent>> OrderValidation
<<initiates>>
<<ProtocolPort>> CheckCustomer (from OrderValidation)
<<initiates>>
<<FlowPort>> reject (from OrderValidation)
<<CompositeData>> OrderDenied

Figure 3-26 Process Components with multiple ports

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

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

3.6.9 Activity Diagram showing the Choreography of a Process Component

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

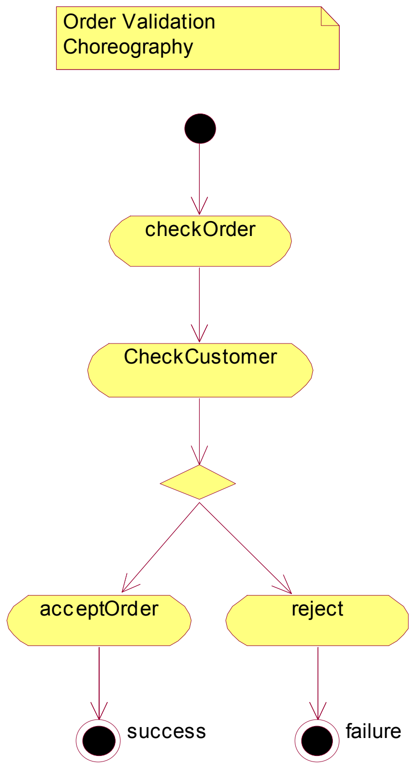


Figure 3-27 Choreography of a Process Component

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

3.6.10 Collaboration Diagram for Process Component Composition

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

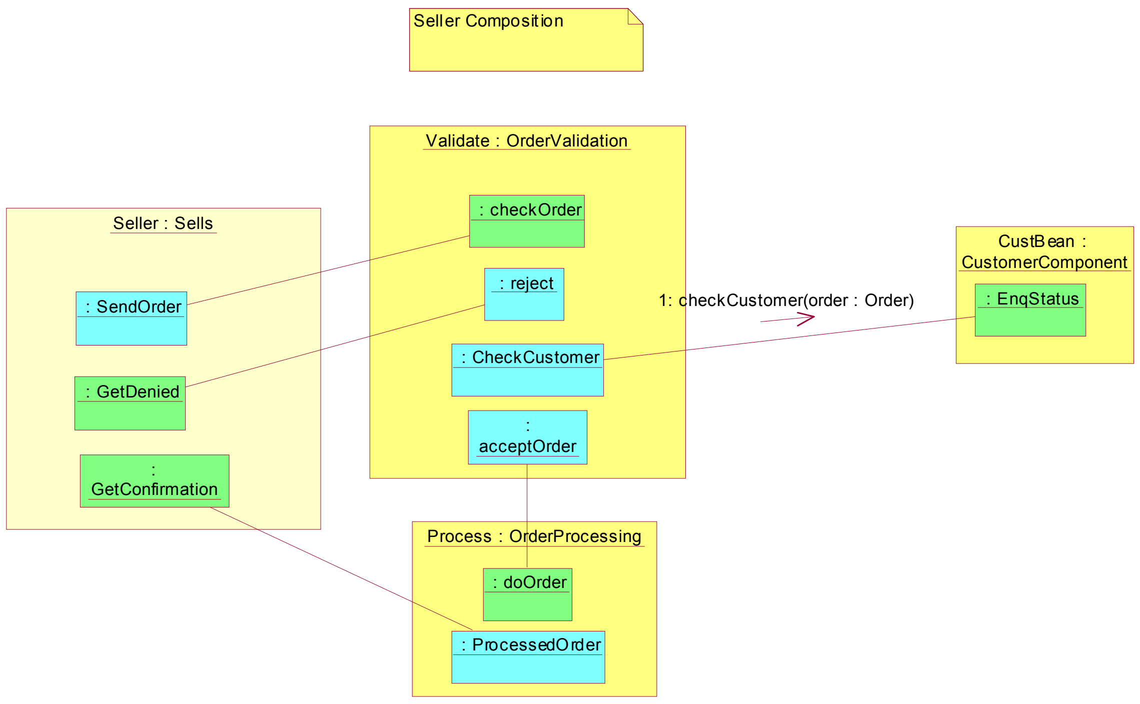


Figure 3-28 Process Component Composition

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

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

The Component Usages are as follows:

- Validate – uses the “OrderValidation” component.
- CustBean – uses the CustomerComponent.
- Process – uses the “OrderProcessing” component (not previously shown).

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

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

To “connect” one port to another we draw an association role (a line representing a Connection) from one port to another. The connected ports must have compatible types and directions. So in this diagram we have made the connections, as listed below.

3.6.10.1 Connections in the example

Table 3-2 Connections

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

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

3.6.10.2 Special note on “proxy” port activities.

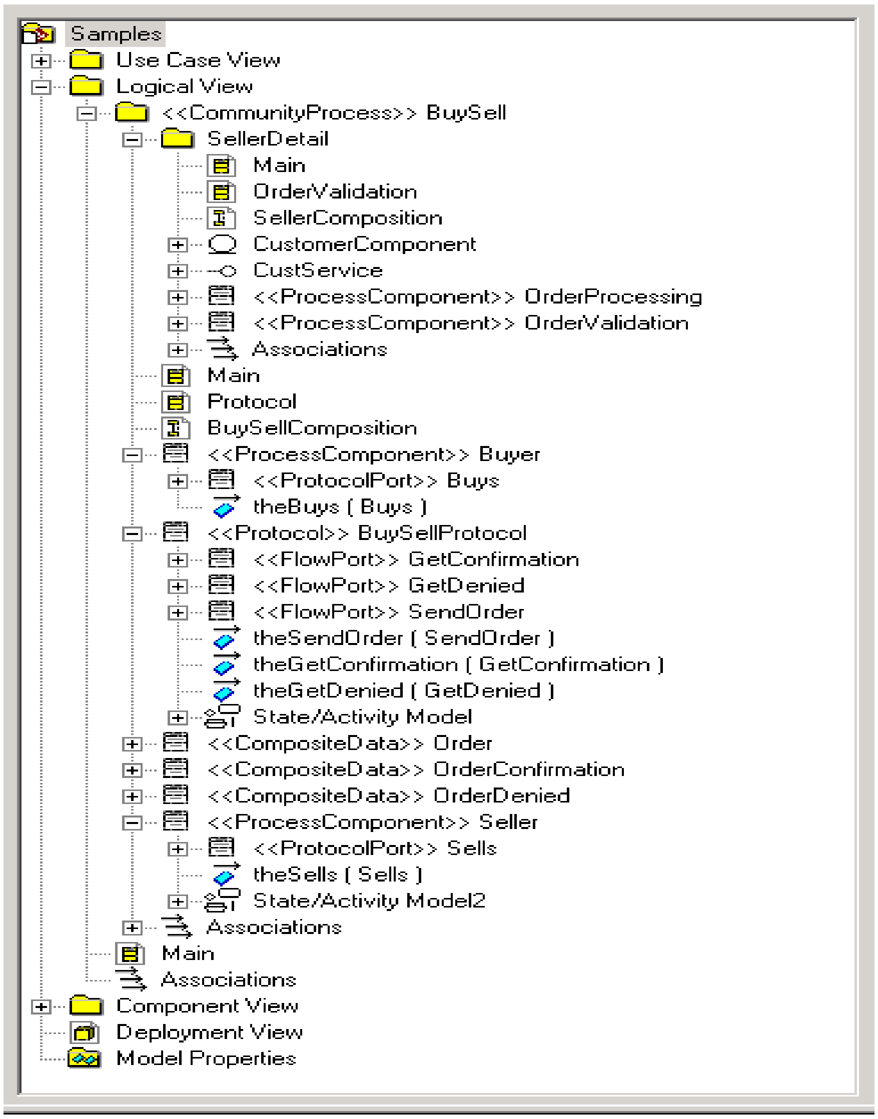
As can be seen from the example, we need to connect the “outside” ports (those on the component being defined) with the “inside” ports (those on the components being used). The PortConnectors for the outside ports are shown without an owning ComponentUsage, while the PortConnectors for the components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

3.6.10.3 Special note on protocols

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

3.6.11 Model Management

While the organizational structure of components is not visible in a diagram, it is visible in tools. The screen shot in Figure 3-29 shows how the example components are organized in the Data Access Technologies’ UML tool. Note how using nested classes (such as Ports being inside of their ProcessComponent) helps to organize the model and keep namespaces separate.



```
Samples
+ Use Case View
- Logical View
    - <<CommunityProcess>> BuySell
        - SellerDetail
              Main
              OrderValidation
              SellerComposition
            + CustomerComponent
            + CustService
            + <<ProcessComponent>> OrderProcessing
            + <<ProcessComponent>> OrderValidation
            + Associations
          Main
          Protocol
          BuySellComposition
        - <<ProcessComponent>> Buyer
            + <<ProtocolPort>> Buys
              theBuys ( Buys )
        - <<Protocol>> BuySellProtocol
            + <<FlowPort>> GetConfirmation
            + <<FlowPort>> GetDenied
            + <<FlowPort>> SendOrder
              theSendOrder ( SendOrder )
              theGetConfirmation ( GetConfirmation )
              theGetDenied ( GetDenied )
            + State/Activity Model
        + <<CompositeData>> Order
        + <<CompositeData>> OrderConfirmation
        + <<CompositeData>> OrderDenied
        - <<ProcessComponent>> Seller
            + <<ProtocolPort>> Sells
              theSells ( Sells )
            + State/Activity Model2
        + Associations
      Main
      Associations
+ Component View
  Deployment View
  Model Properties
```

Figure 3-29 Model Management

3.6.12 *Using the CCA Notation for Component & Protocol Structure*

Figure 3-30 shows the CCA notation being used for the protocol and process component structure, above. Note that as with the UML notation, this is done from an out-of-the-box tool (Component-X®) - the notation is not quite standard CCA yet.

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

.

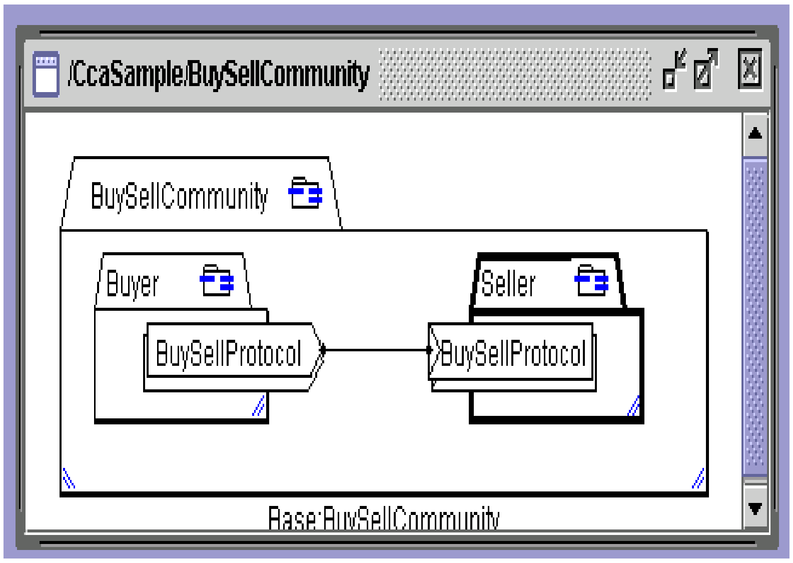


Figure 3-30 Community Process and Protocol

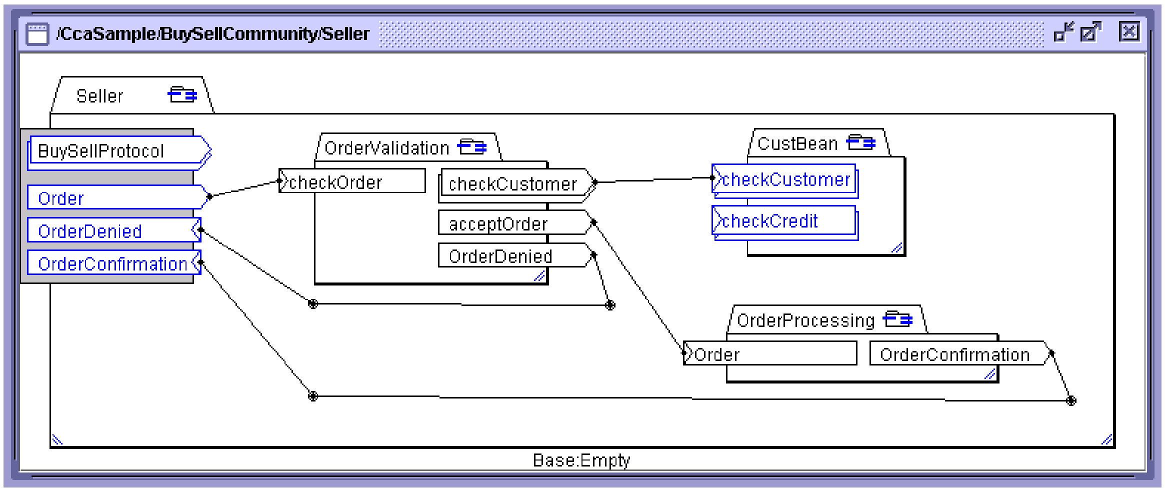


Figure 3-31 Composition in CCA notation

Figure 3-31 shows the seller composition in CCA notation; it is equivalent to the seller collaboration diagram.

Section III - The Entities Model

The Entities model describes a model that may be used to model entity objects that are representations of concepts in the application problem domain and define them as composable components.

Section 3.7 introduces the model and concepts associated with it. Section 3.8 describes different entity viewpoints. Section 3.9 presents the Entity conceptual metamodel.

3.7 Introduction

This section describes the following:

- The Entities model relationship to other models.
- The design concepts incorporated in the Entities model.

3.7.1 Relationship to other parts of ECA

The following paragraphs briefly describe the links to other models in the ECA specification.

3.7.1.1 The Business Process model

The Entities model is used to define a representation of the application domain. Processes operate on this model where the process flow determines that operations should occur on the domain model as a result of inputs from other systems, the occurrence of business events, or the actions of human participants.

The Entities model also provides a root modeling element for identifiable processes. In a business domain a process is also an identifiable concept that has instances with attributes, operations, and relationships. As such, it shares the characteristics of Entity objects and can be operated on the same as entities. A process could be the subject matter of another process.

3.7.1.2 The CCA model

Elements of the Entities model are also characterized as composed components that can be composed into larger components. As components they may be made available for composition of a variety of systems. As composed components, they may be configured from independently created components. The component model determines the unit of composition and the interconnection of interfaces that enables the components to work together.

3.7.1.3 The Events model

The event model defines the integration of systems and components using events to drive the processing. Events may be published or received by entities and processes. Events may be forwarded synchronously or asynchronously. Synchronous events typically will be delivered within the context of the current transaction. Asynchronous events generally will be stored and delivered in the context of a new transaction. The use of events for integration reduces coupling and improves the ease by which a system may be adapted or extended.

The Entities model recognizes the publish and subscribe ports as elements that may be attached to entity components. In addition, it defines the Data Probe port to generate events requested on an ad hoc basis.

3.7.1.4 The Patterns profile

Patterns may be used to replicate frequently occurring entity structures including attributes, relationships, operations, rules, and constraints.

3.7.2 Design Concepts

The entity model reflects the integration of a number of design concepts:

- Composition
- Encapsulation
- Ports
- Identity
- Events
- Domain Modeling
- Entity Role
- Events
- Data Monitoring
- Distributed Computing
- Levels of Coupling

These concepts are each discussed in the paragraphs that follow.

3.7.2.1 Composition

Entities are representations of concepts that exist in the real world or *application problem domain*. The primary purpose of the entity model is to model entities–their relationships, attributes, and methods—and define them as composable components.

The information viewpoint will provide the primary notation for modeling entities, and their attributes and relationships as data. The entities represented in the information viewpoint are then incorporated into objects, described as composable components.

Entities are incorporated into systems where they may be acted upon by processes, interact with other entities, and generate events. Thus entities are components in a larger system. The component relationships of entities to other components is expressed in the composition viewpoint. In this viewpoint entities are components that are composed into larger components.

As a component, an entity may have several different ports. It receives and responds to messages. It may send messages and receive return values, it may generate events or asynchronous messages, and it may receive events or asynchronous messages. In addition, it may accept ad hoc requests to generate messages based on changes in its state.

Entities that represent primary concepts, such as Customer, will often be composed with related entities and value objects as deployable components. So the Customer and Account entities could be composed into one component also containing the Customer Address and Account Entry value objects.

3.7.2.2 Encapsulation

Entity components are intended to be encapsulations of their associated data and functionality. Process Component defined in the CCA specification provides the basic representation of encapsulation. It provides the external interfaces by which these components are linked to other components and composed into larger components. At the same time, it does not define the component implementation.

Data Manager extends this by incorporating Composite Data. Consequently, a data manager contains composite data that describes the state of the component. Data Manager incorporates the composite data and relationships of Entity Data along with methods to operate on the data.

A Data Manager may be implemented as an object. The object has an interface, modeled as a component port, and it has state data that may be accessed through the port. The object may also have other ports. It may have data probe ports to generate messages based on ad hoc requests. It may send asynchronous messages and events. If it has a unique identity (i.e., is an Entity), and is sharable and network accessible, it can receive asynchronous messages and events.

Data Manager comprehends value objects, objects that are passed by value, i.e., by copying the data, not by reference. Consequently, the data structure is exposed when a copy is performed. It is important to distinguish between the value object that has a functional interface, and the state of the value object, the Entity Data, which is passed when a value object is passed as a parameter.

Value objects are not sharable nor network accessible. They cannot receive messages over the network, and they are not sharable because they are always passed by value rather than by reference.

Data Managers may be network accessible or not. A Data Manager may be only accessible by reference to a related entity that is network accessible. For example, an order line item is identifiable but may only be accessible through the order.

An Entity may be a copy of a primary Entity (i.e., a *clone*) for purposes of improving performance. An Entity clone may be a copy of an entity on a client system that is used for interactive operations. Or the clone could be the instantiation of an entity when concurrency control is performed by a database (i.e., the primary entity is in the database). The clone is instantiated with a copy of the entity's state. The primary Entity should be locked when the copy is taken so that its state will not change while operations are being performed on the clone. The clone is not sharable because it should not exist beyond the transaction in which it was created. Its lock on the primary entity will expire when its transaction terminates.

3.7.2.3 Ports

Components interact with their environment through ports. A port has a defined interaction protocol. Ports may send messages, receive messages, or both. A port may be implemented as an object interface (e.g., CORBA or Java interface).

Ports are synchronous or asynchronous. A synchronous port communicates within the context of a transaction. An asynchronous port communicates in a store-and-forward manner so that sending a message occurs in the context of one transaction and receipt of the message then occurs in the context of another transaction.

Ports may communicate with messages or event notices. A message is directed to a specific destination. An event notice is published to the communication infrastructure to be delivered to subscribers—destinations that have expressed interest. The messages and event notices may be communicated synchronously or asynchronously.

All Data Managers will have interface port(s) that represent the interface of the component; these ports may be synchronous, asynchronous, or a combination of both.

3.7.2.4 Identity

Unique identity is introduced on Entity Data and implicitly on Entity with the addition of a Key. A prime key is required to be unique within the extent of the type. In general, identifiable components are passed by reference.

The key may be comprised of one or more attributes of the state of the component, and these elements must be immutable. The key can also have elements that are Foreign Keys of other Entities. A Foreign Key is identified through a relationship with another Entity from which the Foreign Key is derived.

An Entity component has a primary instance (i.e., the location of the master copy of its state). This master copy may be in a database or it may be instantiated as an object/component. Copies of an Entity state may be instantiated in Entity clones. These are not sharable and, in general, should not exist beyond the scope of a single transaction.

Entity components can be “managed.” This property specifies that the extent of all members of a type and its sub-types is known and may be accessed as a set. The key of an identifiable component must be unique within its managed extent. The implementation implication of being managed is that the type will have an extent manager or “home” that will provide query access to the extent and may provide attributes and methods that apply to all members of the extent or the members collectively (e.g., the number of members).

3.7.2.5 Domain Modeling

The first step in modeling a business domain may be to create an information viewpoint. The information viewpoint exposes the Entity Data along with its attributes and relationships. These Entity Data elements will be incorporated into Entity components to define their functionality and interfaces.

In modeling a business domain, business concepts that are uniquely identifiable must be represented by identifiable computational components. For example, an object representing an employee, a purchase order, an office, or a part specification will have a unique identifier that associates the object with the real-world counterpart. As such, a consistent representation of the business will have a single representation of each real-world thing as an identifiable object. While an implementation may replicate such elements for performance or reliability, replicas are still logically a single representation and must be maintained with consistent state if the system is to yield consistent results.

For the most part, the identifiable elements that model the business domain are characterized as Entities. Rules and Processes are also Entities because they have state and are identifiable, but they are computational artifacts that describe activities in which entities are involved.

3.7.2.6 Entity Role

The Entity Role is an important extension to the Entity representation. It may be impractical to design an Entity component to anticipate all circumstances in which an entity may be involved. Each situation may involve different state and behavior. An Entity Role incorporates aspects of an Entity associated with a particular context. Essentially it extends an Entity on an ad hoc basis. The unique identity of an Entity Role is the entity identifier coupled with its context identifier. Consequently, the context must also be represented as an Entity component. For example, a person has the role of an employee as a member of an enterprise (context), or may be a member of a project team. An entity may have many roles as appropriate to the different contexts in which it participates.

An Entity Role is dependent upon the associated parent entity. The association is immutable. If an Entity ceases to exist, all of its roles will also cease to exist. An Entity Role cannot be assigned to another parent Entity.

An Entity Role is not an appropriate representation for such concepts as an organizational position or the specification of a process participant. These concepts may define characteristics of the entities that can be assigned, but should not include characteristics that are unique to a particular Entity when assigned. Consequently, a process participant is an Entity that represents a potential association of a process with an Entity. Different Entities may be assigned to the participation over time. An Entity Role may be assigned to the participation, as an employee may be assigned to participate in a process, and a different employee may be substituted at a later time.

An Entity Role may be a “virtual entity” if it incorporates all of the interface characteristics of the entity it represents. For example, an Entity Role may inherit the interface of its associated Entity, incorporate the interface by inheritance, and incorporate the entity state and behavior by delegation.

3.7.2.7 *Events*

An event represents a change of state in a system that is of interest outside the scope of the component in which it occurs. An event may be defined as a change of state that causes a condition of interest to become true, or an event may be associated with a state transition to a particular state, from a particular state, or from one state to another state. When an event occurs a notice can be generated.

The ability to generate event notices can be designed into a component. The content of the event notice is defined to provide appropriate information about the event. Event notices are published—they are issued to the event communication infrastructure to be received by subscribers. The publisher of an event notice is not expected to be aware of the subscribers, and thus there may be many subscribers or none. Similarly, the subscribers are not aware of the specific sources of event notices to which they subscribe.

The Event Publication and Event Subscription ports provide the complementary interfaces for this publish and subscribe linkage between components. These ports may be defined as operating in synchronous or asynchronous mode.

The mode of a subscriber must match the mode of the receiver for an event notice to be communicated. In synchronous mode, an event notice would be delivered to all subscribers within the context of the transaction in which the event occurred. In asynchronous mode, the event would be delivered in a store-and-forward manner, the event notice would be captured in one transaction and accepted by each subscriber in different transactions.

3.7.2.8 *Data Monitoring*

Data monitoring refers to the ability to ad hoc initiate detection of changes in data in order to initiate desired actions. This capability is an important element of flexibility and modularity of system design. It allows actions to be initiated based on changes in state without explicitly embedding the initiation of those actions in the executable logic that changes the data.

For example, an application may be designed to monitor the price of a commodity to initiate buy or sell orders or alert a customer. It should not be necessary to modify the logic of the commodity tracking system in order to link this monitoring application to price changes.

Similarly, when a system is assembled or extended using components, actions of some components may be dependent on changes in state in other components. By providing the ability to monitor changes in the data of a component, the logic of the component need not be designed to anticipate each specific dependence.

The Data Probe port provides the interface for accepting and removing monitoring requests and for issuing events or messages when the specified events occur in the state of the Entity. A request will define the state of interest, the type of message to be sent, and the message addressee.

3.7.2.9 Distributed Computing

Components that are remotely accessible must be identifiable. Their unique identity is the basis for locating them in the distributed computing environment. It is also the basis for sharing a single representation of the state of the thing being represented.

To support network access, they must have one or more ports that support network access protocols. For example, a network accessible component might have ports synchronous messaging ports implemented as CORBA interfaces, and event subscription and publication ports implemented as JMS (Java Messaging Service) subscriber and publisher interfaces.

Data Managers that are not network accessible will be restricted to being co-located with components that reference them. For example, an order item is uniquely identified within an order, but remote access may be only through interfaces to the containing order.

Relationships require that the participating Entity Data structures are identifiable. At the same time, the Data Manager of an Entity Data structure may not be network accessible. In a distributed computing environment, components that participate in relationships must be either co-located or be network accessible. A relationship cannot be implemented if the members cannot communicate with each other.

While distribution of computing is primarily an implementation issue, the ability for components to be distributed must be considered fairly early in the design. Where Entity components are not network accessible, operations on their containing components will likely reflect indirect access from remote components.

3.7.2.10 Levels of Coupling

The Entity Model anticipates three levels of component coupling: *linked, tightly coupled,* and *loosely coupled*.

Linked coupling refers to components that are co-located in the same address space. These components interact with each other directly, without communicating over a network. As such, they can interact without being network accessible components. Messaging will generally be synchronous, within the scope of a single transaction.

Tightly coupled components are distributed across multiple servers. These components will also interact with synchronous messaging, but messaging will occur over a network. While some messaging between the components may be asynchronous for performance and recoverability considerations, components are tightly coupled if any interactions between them are synchronous.

Loosely coupled components are distributed and only communicate asynchronously, through a messaging infrastructure. Communication is through messages and events. These components might be characterized as enterprise applications. A message or event is issued in the scope of one transaction and accepted by one or more recipients in independent transactions. Messages and events are stored and forwarded. A message is a communicated with a defined recipient, and an event is a communicated (published) with self-declaring recipients (subscribers) unknown to the publisher.

The level of coupling between components has important performance and system flexibility implications. Generally, components should be designed in a level-of-coupling hierarchy so that components that are linked are within components that are tightly coupled, and tightly coupled components are within components that are loosely coupled with each other. This coupling hierarchy should be reflected in the network accessibility property of components and the synchronous vs. asynchronous property of their ports.

3.7.3 Standard UML Facilities

This section briefly describes the standard elements of UML that are incorporated in the model.

Attributes

Composite Data elements define their data elements with attributes. Composite Data elements are incorporated as the data structures of Data Managers, which are specialized to entities. The interfaces to Data Managers provide access to the attributes and will generally have methods by the same name as accessers.

Methods

Methods are specified as in UML. From a component perspective methods, including the attribute accesser methods, are incorporated in the port(s) that receive messages and return a result.

Relationships

Relationships express associations between non-primitive elements. Identifiable, sharable, and network accessible elements can have relationships that extend over a distributed network.

Activity Graphs

Activity Graphs may be used to describe flow of control between elements, although these will be more applicable for describing processes.

State Machines

Changes of state of elements with data may be described with state machines. Publication of events may be defined in terms of state transitions.

Interaction diagrams

Interaction diagrams may be used to describe the flow of control between executable elements.

Object Constraint Language

OCL is used to express conditions for triggers, as well as in other applicable UML elements.

3.8 Entity Viewpoints

The entity model provides elements that appear in different viewpoints. These viewpoints are for different purposes and represent entities differently, using different forms of notation. Two viewpoints of particular interest are presented below: the information viewpoint and the composition viewpoint. Entities also appear in other diagrams, for example, in interaction diagrams as vertical lines and in activity diagrams as swim lanes.

3.8.1 Information Viewpoint

The information viewpoint models Entity Data and their relationships. Entities represent concepts in the problem domain, and relationships represent relationships between the problem domain concepts. The model essentially defines the vocabulary used in discussing the problem domain, and it represents the structure of the objects and databases used to represent the business concepts in the computer.

A model viewed from the information viewpoint is shown below. It includes four Entities: Customer, Address, Account, and Entry. Each of these can be uniquely identified, but Address and Entry are unique within the contexts of Customer and Account, respectively. Consequently, as components, Address and Entry may be specified as not sharable or network accessible. They would be implemented as pass-by-value objects.

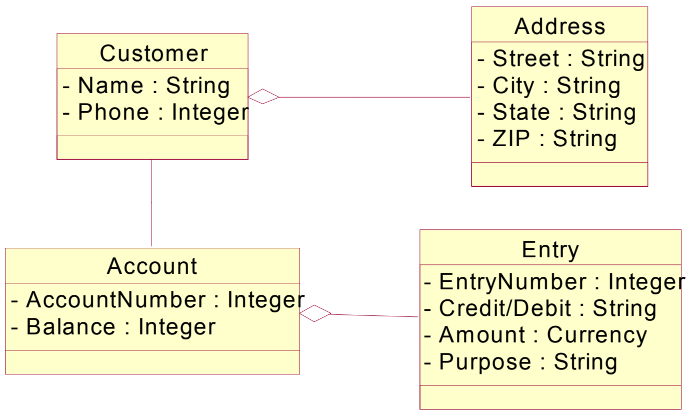


Figure 3-32 Entity Model in the Information Viewpoint

The information viewpoint says nothing about interfaces or object-oriented functionality that may be associated with these Entities. Nor does it define how these objects might be packaged in a composed system. Those aspects are defined by the Entity components that incorporate the Entity Data.

3.8.2 Composition viewpoint

The composition viewpoint describes how the software artifacts are configured as components and compositions of components. The diagram below depicts an Account Composition component, which is composed of Account Entity and Entry Entity components.

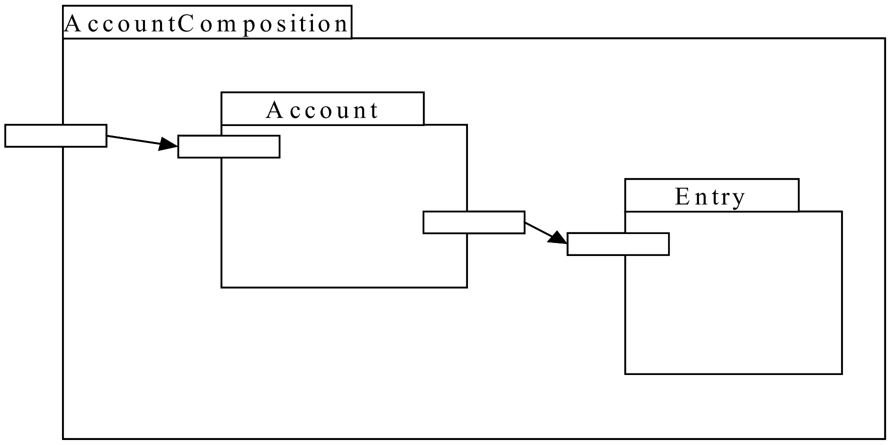


Figure 3-33 Entity Model in the Composition Viewpoint

The Account entity may request attribute values from the Entry object, or, assuming the Entry object is a pass-by-value object, it may pass the Entry object by value. This means it passes a copy of the state of the Entry object, but it retains its reference to the original Entry object for future operations.

The Account and Entry objects are both components used to compose the AccountComponent. However, this could be simply the logical model of the composition. The implementation of the AccountComponent might be primitive, making the Account and Entry objects inseparable, but logically independent.

The ports in this model are interface ports and message-sending ports—they incorporate synchronous messages, typical of messaging with objects. The AccountComposition component may or may not expose the same interfaces as the Account component. It also could expose an interface for the Entry component, but none is specified here.

The composition viewpoint drives consideration of network accessibility and the clustering of objects for composition and distribution.

3.9 Entity Metamodel

This section describes the entity meta-model. This model provides a basis for understanding the modeling concepts and their relationships.

3.9.1 Overview

The diagram, below, depicts the elements to be considered; those that are part of this model specification are highlighted. Central to this model are Data Manager and its specializations; these are the core elements of the Entities model. They encapsulate data and other components, exposing their functionality through ports.

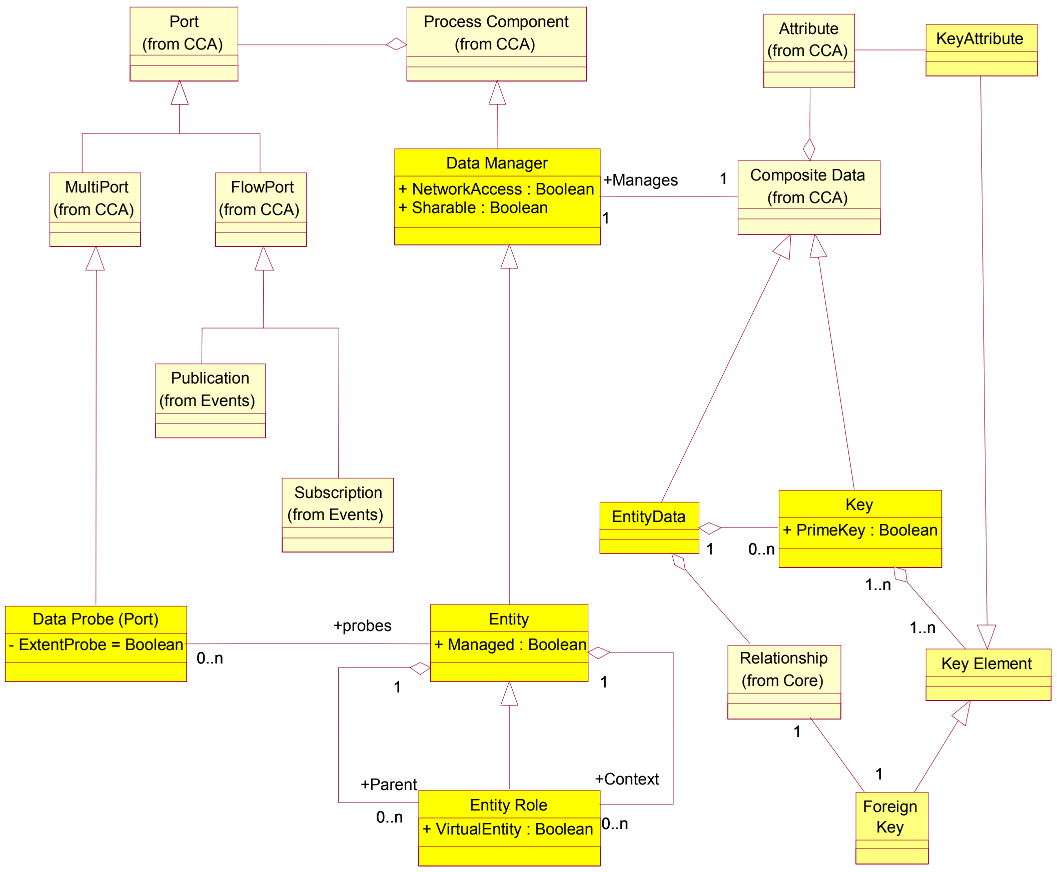


Figure 3-34 Entity Metamodel

Through ports components receive and respond to messages, publish and subscribe to events, and expose state changes response to ad hoc requests to Data Probe ports.

Entities represent the application domain. As components they encapsulate the functionality, state, and relationships of domain concepts. Entity components incorporate Entity Data structures, which are the core elements of the information model.

3.9.2 Entity Package

This section describes the elements of the Entity metamodel in detail.

3.9.2.1 DataManager

Semantics

A Data Manager is a functional component that provides access to and may perform operations on its associated Composite Data (i.e., its state).

The Data Manager defines ports for access to operations on the state data.

Fully Scoped name

EDOC::ECA::Entity::Data Manager

Owned by

Package

Properties

Network Access

A Boolean value that indicates if the Data Manager is intended to be accessible over the network.

Sharable

A Boolean value that indicates if the Data Manager can be shared by multiple transactions/sessions. A Data Manager that is not sharable is either transient or depends on a sharable Data Manager that contains it for persistence. For example, an address may not be sharable (although its state may be passed by value), but it can be persistent by association with a Customer that is sharable.

Related elements

Process Component

Data Manager inherits from Process Component and adds the quality of having associated state.

Composite Data

Composite Data defines the data structure that is encapsulated by the Data Manager.

Entity

Entity specializes Data Manager for representation of identifiable application domain things.

Constraints

N/A

3.9.2.2 EntityData

Semantics

Entity Data is the data structure that represents a concept in the business domain. It is equivalent to an entity in data modeling or a relation in a relational database. In a Data Manager or its specializations, such as Entity, it represents the state of an object.

Entity Data has attributes (from Data Element) and relationships. The information viewpoint is a viewpoint on Entity Data elements.

Fully Scoped name

EDOC::ECA::Entity::EntityData

Owned by

Package

Properties

N/A

Related elements

Composite Data

Entity Data inherits from Composite Data and adds relationships.

Relationship

Describes an association between Entity Data elements.

Data Manager

An Entity Data element is incorporated in a Data Manager that gives it functionality and ports as a component.

Constraints

- Entity Data must have a prime Key that is unique within the extent of the Entity Data type (i.e., the type and all sub-types).
- Entity Data is managed by an Entity Data Manager.

3.9.2.3 Key

Semantics

A Key is a value that may be used to identify a Data Entity for some purpose. Generally, it will be a unique identifier within some context. A Key designated Prime Key = true is the key intended for unique identity of the Data Entity within the extent of the Data Entity type.

A Key is composed of key elements that may be selected attribute values of the associated Data Entity or Foreign Keys. A Foreign Key is the key of a related Data Entity.

Fully Scoped name

EDOC::ECA::Entity::Key

Owned by

Entity Data

Properties

Prime Key

A Boolean value that indicates if the Key is intended to be the primary unique identity of the associated Entity Data type. If so, the value must be unique within the extent of the identifiable type.

Related elements

Composite Data

A Key is a specialization of Composite Data.

Entity Data

A Key describes an identifier of an Entity Data type.

Key Element

A Key Element is one segment of a Key, which is either a reference to an attribute of the associated Data Entity or a reference to the key of an associated Data Entity.

Constraints

- If Key is Prime Key = true, then the value must be unique within the extent of the associated Entity Data type and its sub-types.
- The attributes that are incorporated into the key must be immutable.
- The Key Elements that comprise the key have an immutable sequence.

3.9.2.4 Key Element

Semantics

A Key Element is one segment of a Key, which is either a reference to an attribute of the associated Data Entity or a reference to the key of an associated Data Entity.

Fully Scoped name

EDOC::ECA::Entity::Key Element

Owned by

Key

Properties

N/A

Related elements

Key

The Key in which the Key Element appears.

Key Attribute

A Specialization of Key Element that references an attribute in the associated Entity Data.

Foreign Key

A specialization of Key Element that references the Key of an Entity Data structure that is related to the Entity Data identified by the containing Key.

Constraints

N/A

3.9.2.5 Foreign Key

Semantics

A Foreign Key is a Key Element that is the value of a related Entity Data structure. The subject Entity Data structure derives its identity, in part, from the related Entity Data structure. For example, the line item of an order may be identified uniquely by the line number and the key of the associated order. The Foreign Key element references the relationship in order to identify the related Entity Data that contains the Foreign Key value.

Fully Scoped name

EDOC::ECA::Entity::Foreign Key

Owned by

Key

Properties

N/A.

Related elements

Key Element

Foreign Key is a specialization of Key Element.

Relationship

The associated relationship identifies the Entity Data from which the Foreign Key value is obtained.

Constraints

- If the associated Key has PrimeKey = true, then the relationship used to obtain the Foreign Key value must be immutable.

3.9.2.6 Key Attribute

Semantics

A Key Attribute identifies an attribute of the associated Entity Data that is included as an element of the Entity Data key. The value of the attribute becomes an element of the key of an instance of the Entity Data type.

Fully Scoped name

EDOC::ECA::Entity::Key Attribute

Owned by

Key

Properties

N/A.

Related elements

Key Element

Key Attribute inherits from Key Element.

Attribute

Attribute is the Attribute of the Entity Data structure that is to be incorporated as an element of the containing Key.

Constraints

If the containing Key is designated PrimeKey = true, then the Attribute values that are incorporated into the key must be immutable.

3.9.2.7 Entity

Semantics

An Entity is an object representing something in the real world of the application domain. It incorporates Entity Data that represents the state of the real world thing, and it provides the functionality to encapsulate the Entity Data and provide associated business logic.

An Entity instance has identity derived from the Key of its associated Entity Data.

Entity is the abstract super type of all identifiable application domain elements. This includes Entities that have a collection of rules to operate on the state of related entities. It also includes Entities that incorporate process elements that act on other Entities. The rule set and process specializations introduce additional elements, but have the basic characteristics of being identifiable, having local state (Composite Data) often viewed as their “context,” and having relationships to other Entities that they may act upon.

If an Entity is managed, all instances of the type and its sub-types are known, each instance has unique identity, and the type can have operations and attributes associated with the extent (i.e., applicable to all instances). This is typically implemented as a type manager or “home” object that represents the extent.

Fully Scoped name

EDOC::ECA::Entity::Entity

Owned by

Package

Properties

In the list below only Managed is introduced as a property by Entity, but NetworkAccess and Sharable, inherited from Data Manager, are also discussed to clarify the implications.

Managed

A Boolean value that indicates if the Entity type is *managed*. If it is managed, then the implementation provides a mechanism for accessing the extent of all instances of the type and its sub-types and may provide a mechanism for dynamically applying rules to all instances. This typically is implemented as a “home” or “type manager.”

NetworkAccessible

A Boolean value that indicates if the Entity is expected to be accessed over the network. This implies that it has a network interface (e.g., CORBA IDL). An Entity that is not NetworkAccessible can only be accessed over the network through an associated Entity that is NetworkAccessible.

Sharable

A Boolean value that indicates if the Entity can be shared by multiple, concurrent transactions or users. A Sharable Entity will enforce controls to serialize access by concurrent transactions.

An Entity that is not sharable may be instantiated for use by a particular user or transaction. It generally contains a copy of the primary Entity Data instance representing the real world thing. The primary Entity Data instance may be in a database and the copy is created to perform operations on the Entity Data. Alternatively, the Entity Data may be managed by an Entity that is sharable, but the copy is created so that processing can be localized on another server. In either case, it would be expected that the primary Entity Data would be locked when the copy is taken and released when the copy is deleted. Changes to the copy would likely be applied to the primary instance prior to removing the lock.

Entities that are not sharable may also be implemented as value objects, which are always passed by value over the network. While they may have unique identity by association with an identifiable Entity, they may not have a key that reflects this unique identity and their Entity Data does not carry its unique identity when passed by value.

An Entity that is sharable is expected to be persistent. An Entity that is not sharable may be persistent if it is incorporated in the state of a sharable Entity.

Related elements

DataManager

Entity inherits from DataManager and adds the requirement that its associated Composite Data is Entity Data. It also adds the ability to accept Data Probes and the ability to be Managed.

Entity Role

Entity Role inherits from Entity as a specialized representation of an Entity in a particular context. The Entity Role contains Entity Data that is associated with the parent Entity in the particular context. Entity Role is associated with another Entity that represents the context in which it applies. Thus the parent Entity might be a person, the Entity Role might be the person as an employee, and the context entity might be the employer.

An Entity may have many Entity Roles. Each Entity Role defines characteristics of the Entity in a particular context, such as person in the role of an employee within a corporation. An Entity may be the context for many Role Entities as a corporation is the context of many employees.

Data Probe

A Data Probe port is associated with an Entity that accepts requests to detect changes in the internal state of the Entity and forwards messages or events when the states of interest become true.

Constraints

- An Entity manages Entity Data, which may have a key and relationships.
- A managed Entity must have a Primary Key.
- A network Accessible Entity must have a Primary Key.
- An Entity that is Sharable will serialize concurrent transactions that attempt to access its data.

3.9.2.8 Entity Role

Semantics

An Entity Role extends its parent Entity for participation in a particular context. An Entity may have a number of associated Entity Roles reflecting participation in multiple contexts. The Entity might have several Entity Roles of the same type at the same time, but each should be associated with a different context.

The context of an Entity Role is also represented by an Entity. The context could be a corporation where the parent is a person and the Entity Role is an employee. A context may have many entity roles of the same type or different types representing participation of different parent Entities for different purposes.

Fully Scoped name

EDOC::ECA::Entity::Entity Role

Owned by

Entity (context)

Properties

VirtualEntity

A Boolean value that indicates if the Entity Role incorporates and extends the primary interface of the parent Entity it represents (i.e., it can be used in place of the primary Entity).

Related elements

Entity

- Inheritance—Entity Role inherits from Entity such that it functions as an entity but it derives its unique identity from the Entity it represents (i.e., a Foreign Key).
- Context association—An Entity Role represents an Entity in a particular context. This association defines the context.
- Parent association—An Entity Role represents an entity in a particular context. This association defines the parent Entity being represented.

Constraints

The parent entity of an entity role cannot be dynamically changed.

3.9.2.9 DataProbe

Semantics

A Data Probe port is associated with an Entity and accepts ad hoc requests to detect changes in the internal state of the Entity. The Data Probe then forwards messages or events when the states of interest become true until the request is removed. A Data Probe may serve many requests concurrently, producing various messages or events when the appropriate states occur.

Fully Scoped name

EDOC::ECA::Entity::Data Probe

Owned by

Entity

Properties

ExtentProbe

ExtentProbe = true indicates that requests apply to the extent of the associated entity as opposed to a particular instance. In implementation, an ExtentProbe would be associated with a “home” or “type manager.”

Related elements

Multi Port

Data Probe inherits from Multi Port.

Entity

The Entity that will accept probe requests.

Constraints

- DataProbes only emit messages (i.e., output only).
- DataProbe can only attach to an Entity with Managed = true.

Section IV - The Events Model

The Events model describes a model that may be used on their own, or in combination with the other EDOC elements, to model event driven systems.

3.10 Rationale

3.10.1 Introduction

Event driven computing is becoming the preferred distributed computing paradigm in many enterprises and in many collaborations between enterprises.

Event driven computing combines two kinds of loosely coupled architectures:

- Event driven process architecture. This is a loosely coupled process architecture where the activities are not sequenced in traditional workflow fashion. Rather each participant in the process has autonomous responsibilities and performs those responsibilities on the basis of loosely coupled notifications, (in the supply chain world a.k.a. business signals).
- Publish and subscribe information distribution architecture. Publish and Subscribe is a loosely coupled mechanism for getting information from publishers to subscribers, while keeping the two independent of each other. Publish and subscribe is often implemented as loosely coupled, distributed components that communicate with each other through asynchronous messaging.

In event driven computing the most important aspect of the business process is the events that happen during its execution, and the most important part of the component-to-component communication is the notification of such events from the component that made them happen to all the components that need to react to them.

In ECA we support both the definition of loosely coupled event-driven business processes, and the loosely coupled publish and subscribe communication between distributed components.

Neither the business world, nor the computing world, however, applies only one paradigm to their problem space. Businesses use a combination of loosely coupled and tightly coupled business processes and computing solutions deploy a combination of loosely coupled and tightly coupled styles of communication and interaction between distributed components.

This document describes in detail the event-driven flavor of loosely coupled business and systems models, and also illustrates how such models can co-habit with more tightly coupled models.

An ECA based business process can be defined as event driven for some of its steps and workflow or request/response driven for others. Likewise, distributed components in the ECA component model can be configured to communicate with each other in a mixture of publish-and-subscribe, asynchronous Point-to-Point, and client-server remote invocation styles.

This document focuses on the purely event driven paradigm, and covers the following topics:

- Design rationale
- Event driven business model
- Event driven computing
- Event driven business computing
- Publish and subscribe
- Key concepts of event driven business and system models
- Metamodel for specifying event driven business systems
- Relationship to other ECA models
- Relationship to other paradigms
- Applicability and leverage of event driven models

3.10.2 Overall Design Rationale

This model is based on the following design principles:

- Alignment with the BOI roadmap (BOM/98-12-04) with respect to business process, business entity, business event, and business rule.
- The event as a central rather than peripheral concept.
- Business Processes should be loosely coupled:
  - Autonomy of participants in a business process
  - Distinction between process and entity
  - Clear separation of business logic (i.e., rules from business execution (i.e., the action taken once rules have been resolved)).
- Information distribution should be loosely coupled
  - Use of Publish and Subscribe rather than point-to-point
  - Ubiquitous event notification

    - Asynchronous computing
    - Shared information model

- Loose coupling of the Events model with the Business Process model, Entities model, and component model
- Re-usability of paradigm
    - Recursive use of event notifications
- Applicability under multiple paradigms
    - The Events model is intended to support both business process modeling and EAI.
    - The proposed model is intended for either tightly coupled client/server or peer-to-peer computing, or loosely coupled event-driven computing, or combinations of both.

3.10.3 Concepts

3.10.3.1 Event based business model

An event based business model is driven by business events. Whenever a business event happens anywhere in the enterprise, some person or thing, somewhere, reacts to it by taking some action. Business rules determine what event leads to what action. Usually the action is a business activity that changes the state of one or more business entities. Every state change to an Entity constitutes a new business event, to which, in turn, some other person or thing, somewhere else, reacts by taking some action.

The main concepts in event driven business models are the business entity, business event, business process, business activity, and business rule.

This continuous, cyclical view of the interaction between these five business concepts can be depicted as follows:

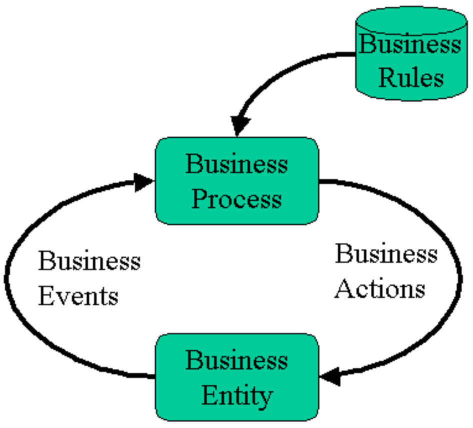


Figure 3-36 Event Based Business Modeling

3.10.3.2 Event Driven Computing

Event driven computing is a computing paradigm where interaction among components is based on notification of what happened, as opposed to instructions of what should happen.

"What happened" is reflected as events. The communication that the event happened is reflected as notifications. The reaction to the notification (or indirectly to the event) is reflected as activities.

Two important layers provide loose coupling between event, notification, and activity.

The events are decoupled from the act of notification by configurable subscriptions.

The act of notification is decoupled from the activity by configurable notification rules.

Event driven computing is a very flexible, yet powerful architecture for enterprise distributed object computing. The main architectural principle is that individual components are kept as autonomous as possible, and that the loose coupling and configuarability enable rapid reconfiguration of the system to meet changing business model requirements such as mergers, outsourcing, and business re-engineering. Under event driven enterprise computing all business entities are self-contained, and typically do not directly change each other's state.

3.10.3.3 Event Driven Business Computing

Event driven business computing is a paradigm that executes business processes by capturing events that happen in the enterprise, notifying the appropriate other parties in the enterprise or outside the enterprise, and reacting to such notifications.

Business processes are configured with a set of subscriptions, and a set of notification rules that determine what activity to start (or end) based on each notification.

Business Entities are the people, products, and other business resources and artifacts that business activities operate on. When actions are performed on Business Entities, Business Events happen. All Business Entities are capable of notifying the world of events that happen to them.

Business Processes that are capable of subscribing to such event notifications are called EventBasedProcesses. They assign notifications to activities based on a set of Notification Rules.

3.10.3.4 Publish and Subscribe

In a Publish and Subscribe information distribution model, publishers publish information, and subscribers subscribe to information. Publishing simply means make the information openly available for consumption. Subscribing simply means expressing an interest in the information and consuming it when it gets delivered. The information is transferred from Publisher to Subscriber ‘automatically,’ usually through the use of asynchronous message middleware. Publishers do not know which subscribers will receive their data, and subscribers do not know where the information comes from. The information, however, describes the state of a process or an entity that is of interest to both publisher and subscriber, and both parties share the information model that describes these states (and state changes).

3.10.4 Key Concepts of Event Driven Business and System Models

3.10.4.1 EventBasedProcess

This is a concept introduced by this ECA Events model, but based on the Choreography element in the ECA component model.

EventBasedProcesses are identifiable series of activities that change states of business entities, thereby causing business events. For example, the activities in the Shipping process may cause allocation events against the Inventory Entity, and pick, pack, and ship events against the Shipment Entity.

3.10.4.2 Entity

This is a concept from the Entities model.

Business Entities are representations of entities of significance to the business, identifiable by an ID, operated on during business process execution, and characterized by having a lifecycle expressed as a set of entity states. Examples are Customer, Purchase Order, Product, and Payment. In the Events model, we use the supertype of Entity, DataManager as the managers of the data behind an Entity. An EventBasedDataManager is capable of publishing information about all changes to the data it manages. Because an EventBasedDataManager is a kind of EventBasedProcess, it can also publish information about state changes in its internal process.

3.10.4.3 BusinessEvent

This is a concept introduced by this ECA Events model.

BusinessEvents are state changes whose occurrence is of significance to the execution of business processes. Typically business events reflect state changes in Business Entities. These can be thought of as entity events. Examples are the approval of a Purchase Order, or Receipt of a Payment. A more indirect type of business event is a state change to a business process or to a collaboration between two business processes. These are called ProcessEvents.

3.10.4.4 Notification

This is a concept introduced by this ECA Events model. This is a concept only, it is not represented by a specific element in the Events model. It is implemented using the dataflow part of the Business Process model.

A notification is a triggered dataflow between two roles, or between two components. The trigger that causes the notification can be 'manual,' or timed, or it can be due to the fact that an event has happened. When triggered by an event, it is called an event notification. Event notification, too, is just a concept, and not modeled explicitly.

The notification is always one-way only. The source of the notification is usually an Entity, but can also be an EventBasedProcess. The destination is usually an EventBasedProcess.

A notification can be thought of as the delivery of a set of data from a publisher to a subscriber. The data delivered is a PubSubNotice. A PubSubNotice is just a set of data, it is immutable, and it does not have any behavior of its own. There is no implication in the PubSubNotice as to what the recipient is going to do when it receives the PubSubNotice.An EventNotice is a special kind of PubSubNotice.

All business events are associated with an EventNotice and the corresponding notification will take place whenever the business event happens successfully.

Similarly, when a business event is supposed to have happened but didn't, 'failure' notifications will take place.

An EventNotice always conveys the following information:

- EventBasedProcess or entity the event happened against,
- trigger that caused it,
- identification of the before state,
- after state,

- change between the two states.

3.10.4.5 *Publisher*

This is a concept introduced by this ECA events model.

A publisher is a component that provides PubSubNotices.

3.10.4.6 *Subscriber*

This is a concept introduced by this ECA Events model.

A subscriber is a role or component that holds subscriptions to one or more PubSubNotices.

3.10.4.7 *Subscription*

This is a concept introduced by this ECA Events model.

A subscription establishes a flow of PubSubNotices to the subscriber. A subscription identifies the type of EventNotice (e.g., the kind of event you want to be notified about). A subscription may additionally have a SubscriptionClause associated. The SubscriptionClause functions as a filter much like a where-clause on the content of the notification.

3.10.4.8 *NotificationRule*

This is a concept introduced by this ECA Events model.

NotificationRules are rules that govern the execution of (part of) an EventBasedProcess. A NotificationRule is a mapping from a BusinessNotification to an activity, optionally guarded by an EventCondition. An EventCondition is a dependency on the receipt of additional, related PubSubNotices.

3.10.5 *Event and Notification based Interaction Models*

The basic building blocks are the EventBasedProcess and the Entity, as shown in Figure 3-37. The two are ‘wired together’ by a flow of actions from process to entity, and by a flow of EventNotices from entity to process. In a component framework, therefore, EventBasedProcesses have EventNotices inflow and action outflow, and Entities have action inflow and EventNotice outflow. A messaging infrastructure manages the delivery of EventNotices from entities to processes. The actions too, incidentally, can be implemented via a messaging infrastructure, but the corresponding messages are usually point-to-point.

This means that we can create CCA EventBasedProcess components and CCA event-based Entity components if we can model:

- An EventBasedProcess as a set of Notification Rules of the type notification/condition/activity. (This is the event-driven equivalent of the commonly known even/condition/action rule.)
- An event-based Entity as a set of action/state/event causalities.

The connection from EventBasedProcess to Entity is governed by a configurable mapping of notification to action, namely the notification rule.

The connection from Entity to EventBasedProcess is governed by a configurable set of subscriptions.

With these building blocks we can model a number of event-based interactions. And by reconfiguring the Notification Rules and/or the Subscriptions, we can easily re-engineer the business process and its execution in the system.

The very simplest model is a single process affecting a single entity, but this is not very interesting.

The simplest model of interest is a single process affecting multiple entities.

A slightly more complex interaction is process-to-process notifications. This model is used in supply chain models, a.k.a. business signal.

Another flavor of interaction is the delegation of the responsibility to deal with notifications. This model is used in EAI integration where legacy applications can be "wrapped" behind publishers and subscribers of notifications.

These three flavors map to three kinds of interaction in the component model:

- interaction between a master and slave component,
- interaction between two peer components, and
- interaction between the boundary of a component and its subcomponents.

Yet another kind of interaction that can also be based on events and notifications is a collaboration between processes. This model is used often in b2b interactions. Even web services can be implemented using event concepts and loosely coupled messaging.

3.10.5.1 *Intra Process Event Notification*

The simplest model is a single process affecting multiple entities. This can be modeled pictorially as shown in Figure 3-37.

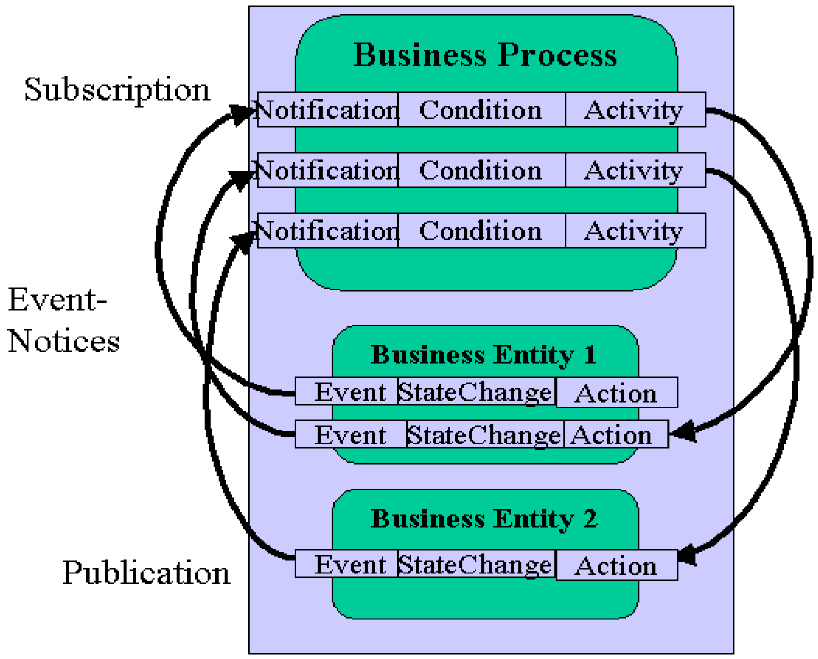


Figure 3-37 Intra Process Event Notification

This corresponds to interaction between a master and a set of slave components. The process has the logic to evaluate notifications and invokes actions on the entities.

3.10.5.2 *Cross Process Event Notification*

A picture of loosely coupled cross process notification:

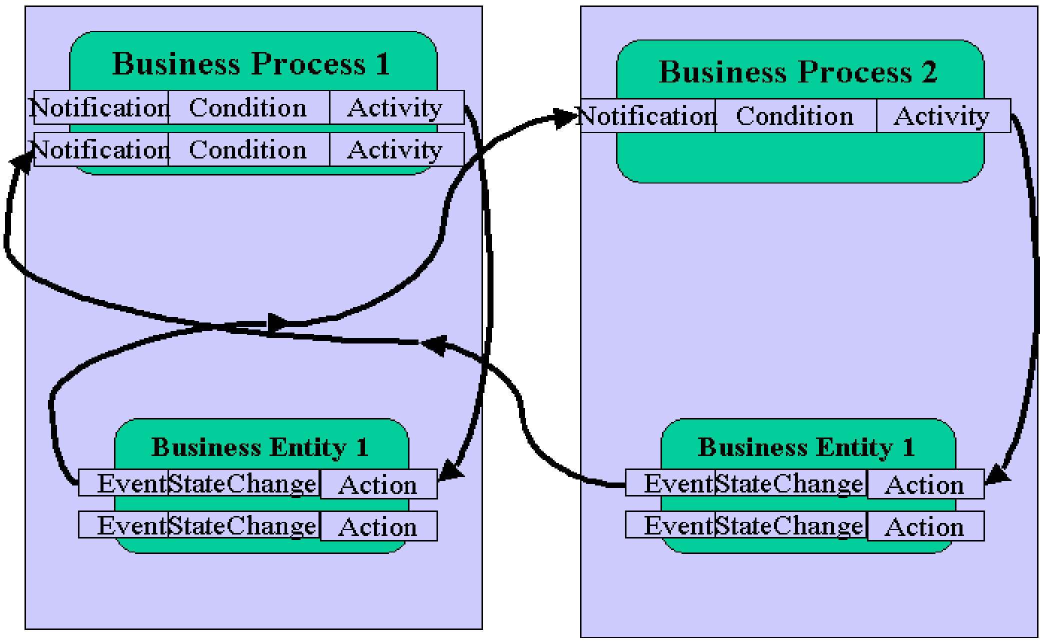


Figure 3-38 Cross Process Event Notification

This corresponds to interaction between two peer components.

3.10.5.3 Delegation

Delegation is passing on of a responsibility. Relative to the event driven model, delegation is the passing of the business notification to another process, for it to resolve, typically a sub process. There is a distinct expectation that the business activity will happen, but it will happen as part of the sub process, not in the main process. However, to the outside processes it will appear as if the main process performed the business activity, and any events will look like they happened in the main process and any notifications will come from the main process.

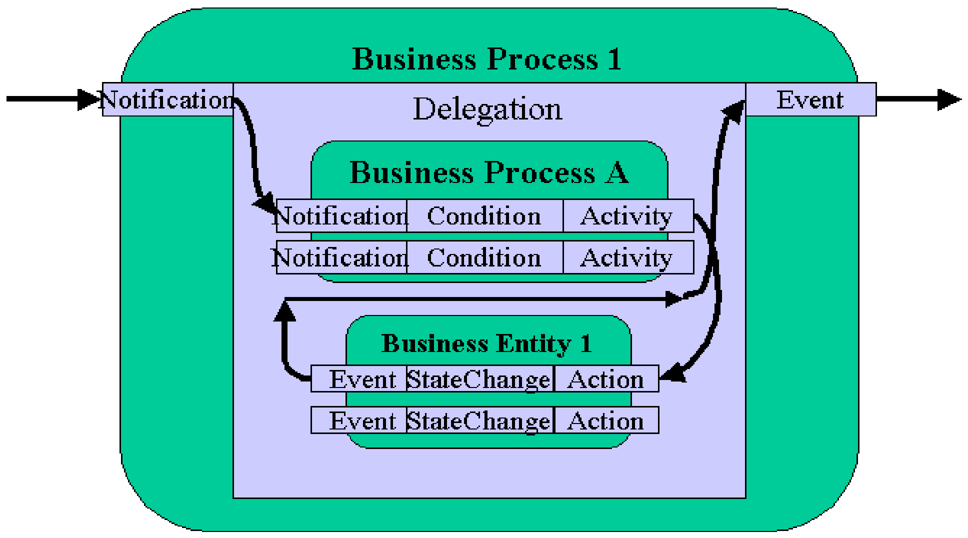


Figure 3-39 Delegation

In the component model this is the interaction between the boundary of a component and its subcomponents.

3.10.6 Leveraging Event Based Models

3.10.6.1 Business Event Types

A variety of standard event types enable a rich set of event-based scenarios.

Success events

A success event is the ‘normal’ event. It reflects the successful execution of an action on an entity or the successful initiation or completion of an activity within the process.

Failure events

A failure event is a type of ‘exception’ event. It reflects that an action on an entity was attempted but failed, or that the initiation of an activity failed, or that an activity was forced to terminate unsuccessfully. In programming languages this is the equivalent of ‘raising an exception.’

TimeOut-Events

This is one of the most useful events for management. A TimeOut-event is an abstract event that reflects that something should have happened within a certain time period, but didn't. Typically this would be something like 'shipment was scheduled but did not happen' within the allotted time. This can be generated based on an overdue condition relative to a scheduled time.

Mutual exclusion events

This type of event signifies that a given event that might be expected according to the business process did not happen due to another alternative event happening. This may be due to the process calling for a mutually exclusive choice between two parallel events, or based on the occurrence of an event that normally happens after the event in question, indicating that an event was 'skipped.'

Data change events

These are useful for replication of data from one place to another. Whenever the source data changes, events are generated, even if the change in data is not considered an event in an entity life cycle sense.

Timed notifications

This is in some sense the simplest kind of notification; it is simply an alarm clock or planning calendar. You can schedule notifications based on a schedule of trigger times. The event, in some sense, is the clock reaching the scheduled time. The notification is usually about the state of something as per that time, or in some cases it could be the timed release of a number of accumulated event notifications.

3.10.6.2 Event Algebra

Events may be ANDed/ ORed, included, excluded, to create new event types. For instance:

CreditApproved event, and shipmentReady event may be ANDed to releaseApproved event.

OrderApproved event and NOT licenseDenied event may be ANDed to shipmentReleased event.

OrderShipped event, and NOT shipmentInvoiced event may be ANDed to invoice exception event.

OrderShipped event and orderCanceled event may be ORed to produce an orderClosed event.

Such event algebra is performed by value-added event agents. They take event notifications as their input and produce value added event notifications as their output.

Such an agent could also be turning event notifications into time-released notifications.

3.10.6.3 Management by Exception

One of the most important ways to leverage event driven computing is to manage by exception notifications. If the business model defines all the events that should occur in the normal course of business, then intelligent agents can be set up to track the progress of each process instance and issue notifications whenever something happened too late or didn't happen at all. These agents would issue timeout-event notifications, mutual exclusion event notifications, and other exception notifications.

Event notifications can also be used to monitor workloads and to give input for rebalancing of loads within a process.

3.11 Metamodel

This is a meta-model for event-driven business computing, specifying the concepts described above. The model consists of two packages:

- Publish and Subscribe Package
- Event Package

These two packages are described in detail below, but first we show two views across the packages:

- Process View (showing how a Business Process produces and reacts to events)
- Entity View (showing how Business Entities produce and react to events)

We also show both packages and both views together in a full overview diagram of the metamodel for the Events model.

3.11.1 Business Process View

This is an overview of the business process aspect of event-driven business computing. The yellow (shaded) elements are directly part of the business process view. The white elements belong to other views and provide the context for this view (see Figure 3-40).

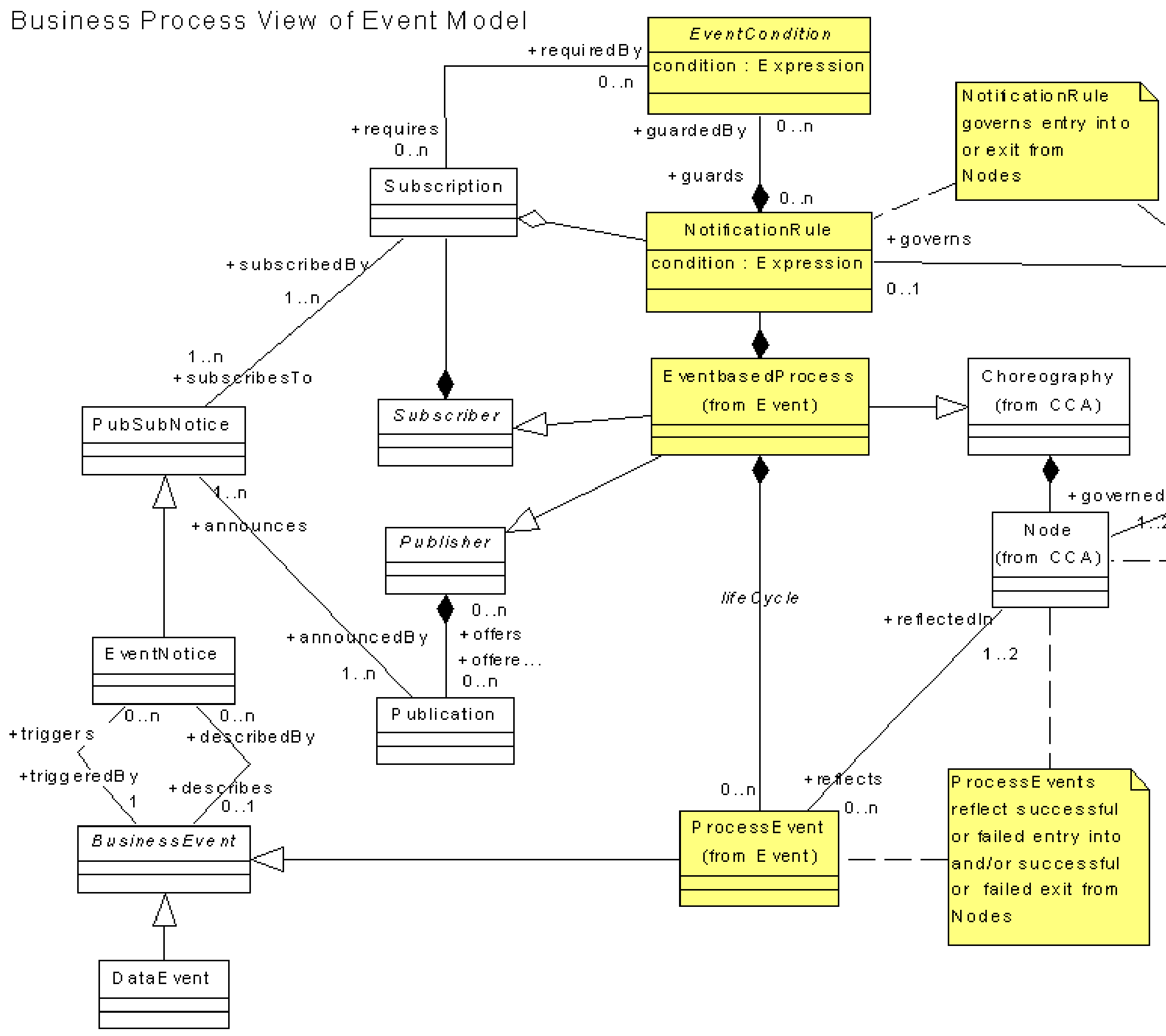


Figure 3-40 Business Process View of metamodel

An EventBasedProcess is a specialized choreography. A choreography (from the CCA Model) is a set of Nodes (States and PortUsages) and the Connections between them. An EventBasedProcess generates ProcessEvents upon successful or failed entry into or exit from its Nodes. A ProcessEvent is a kind of BusinessEvent. An EventBasedProcess is a Publisher and will publish EventNotices for each of its ProcessEvents. An EventBasedProcess is also a Subscriber and will hold subscriptions to PubSubNotices, specifically EventNotices from other processes and from entities.

The NotificationRule is the loose coupling between the receipt of an EventNotice and entry into or exit from a Node. One or more EventConditions may guard the NotitificationRule. An EventCondition requires the receipt of an additional EventNotice, governed by another subscription.

3.11.2 Entity View

This is an overview of the entity aspect of event-driven business computing. The yellow (shaded) elements are directly part of the entity view. The white elements belong to other views and provide the context for this view.

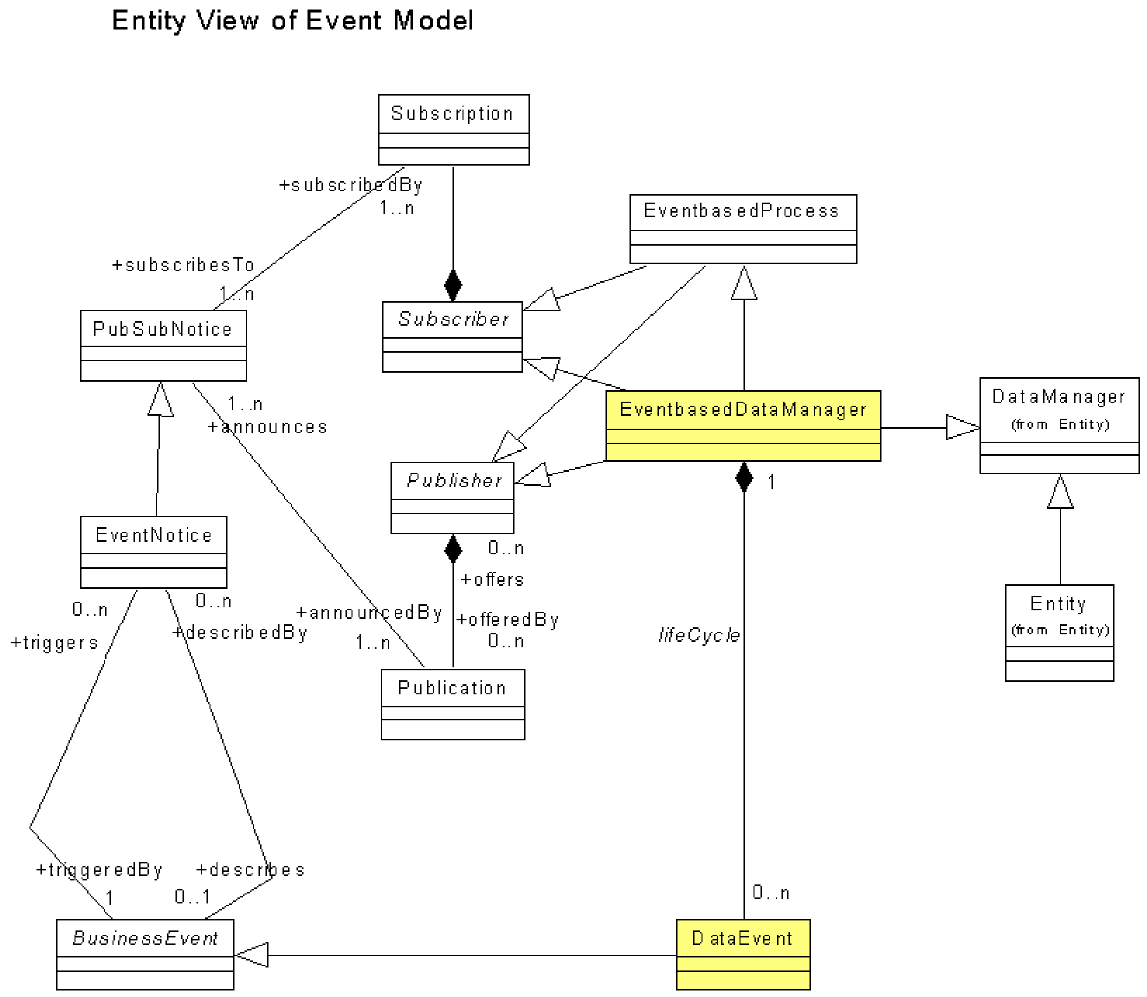


Figure 3-41 Entity View of metamodel

In the Entities model Entity is a kind of DataManager. Further, a DataManager is a kind of Choreography. An EventBasedDataManager is a special DataManager that generates DataEvents each time its data changes. DataEvents are a kind of BusinessEvent. Since a DataManager is also a kind of Choreography, it can also generate ProcessEvents about its own internal choreography.

3.11.3 Whole Event Model

The following is a diagram of the whole metamodel for the Events model. The yellow (shaded) elements are directly part of the metamodel, and will be described in detail below, divided into two packages: Publish and Subscribe, and Event. The white elements belong to other models and provide the context for this view.

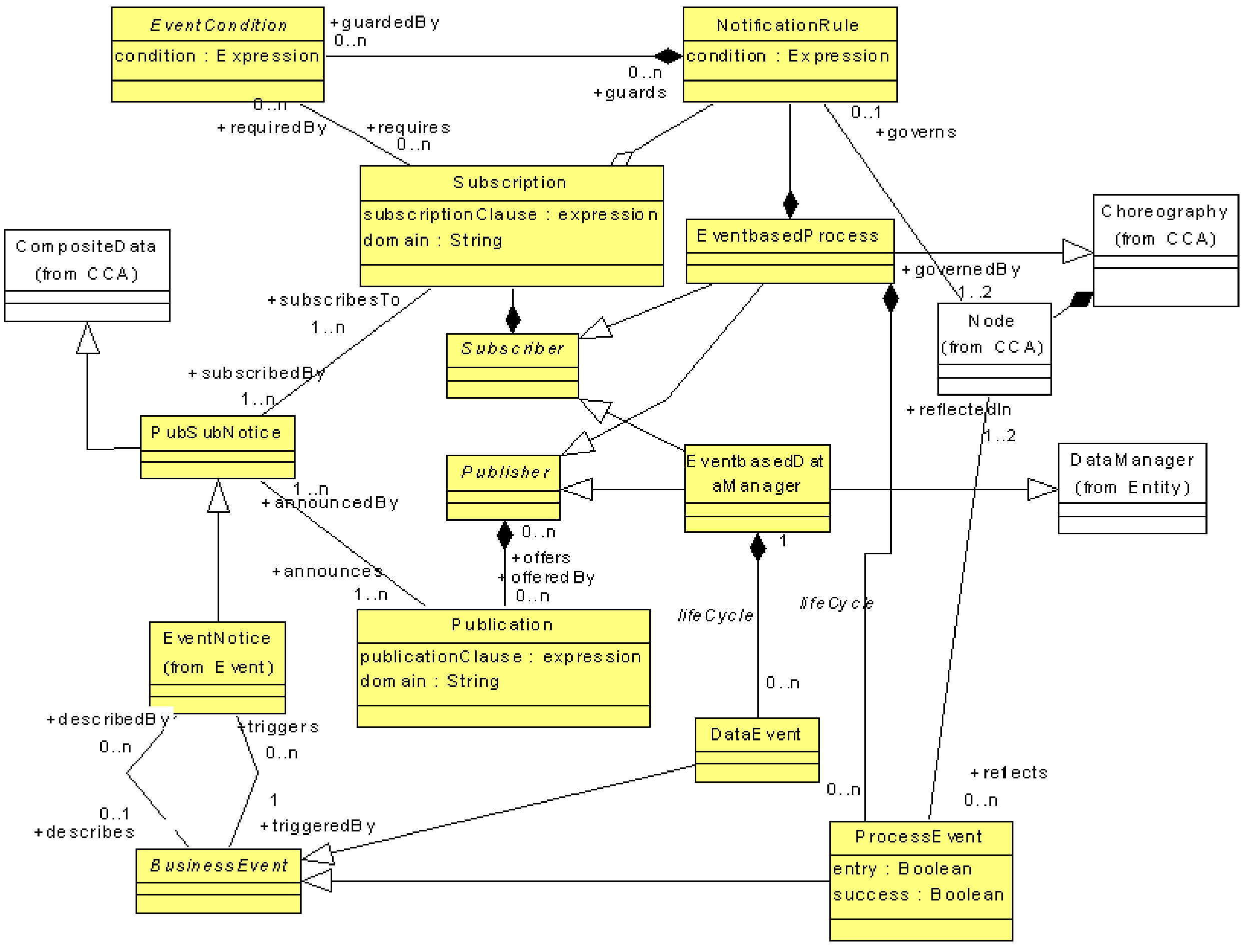


Figure 3-42 Complete Metamodel for Event Modeling

3.11.4 Publish and Subscribe Package

This is an overview of the publish and subscribe aspect of event-driven business computing. The yellow (shaded) elements are directly part of the publish and subscribe package. Each of them will be described in detail below. The white elements belong to other views and provide the context for this view.

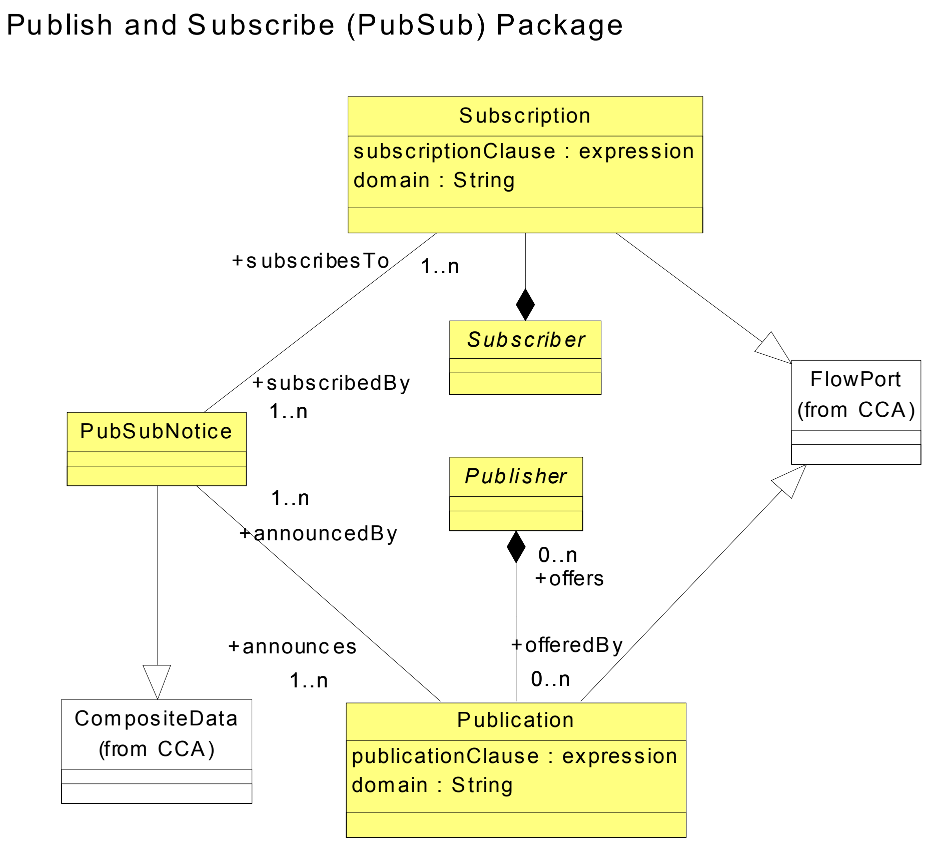


Figure 3-43 Metamodel of event notification view

A publisher is a component that offers a list of publications, and produces (publishes) PubSubNotices accordingly. Publication is the commitment to send PubSubNotice. PubSubNotice is the data structure in which the PubSubNotice instances will be published.

EventNotice is a kind of PubSubNotice.

A subscriber is a component that holds Subscriptions and receives PubSubNotices accordingly. Subscription is the loose coupling between the sending of the notice and the receipt of the notice. A subscriptionClause determines whether the subscriber gets notified or not.

Notification is the sending of a PubSubNotice from the Publisher to the Subscriber when an event happens within the Publisher. This is usually handled by middleware, and publisher and subscriber are loosely coupled and anonymous relative to each other.

3.11.4.1 Publisher

Semantics

A publisher is a component that exposes a list of publications and produces PubSubNotices accordingly.

Fully Scoped name

EDOC::CCA::Event:: Publisher

Owned By

None

Properties

None

Related elements

Publication

Publisher *offers* one or more Publications.

Constraints

None

3.11.4.2 Publication

Semantics

A Publication is a declaration of capability and intent to produce a PubSubNotice.

Fully Scoped Name

EDOC::CCA::Event:: Publication

Owned By

Publisher

Properties

publicationClause

Expression based on attributes of PubSubNotice, describing the instance subset that will be produced according to this publication.

domain

A domain in which the PubSubNotices for this publication will be produced.

Related Elements

Publisher

A Publication is *offeredBy* exactly one Publisher.

PubSubNotice

A Publication *announces* one or more PubSubNotices.

FlowPort

A Publication *Inherits* from FlowPort as per the Component Model.

Constraints

PublicationClause Expression is constrained to the values of the attributes of the associated EventNotice.

3.11.4.3 Subscriber

Semantics

A subscriber is a role or component that exposes a list of subscriptions and consumes PubSubNotices accordingly.

Fully Scoped Name

EDOC::CCA::Event:: Subscriber

Owned By

None

Properties

None

Related elements

Subscription

A Subscriber *holds* one or more Subscriptions.

Constraints

None

3.11.4.4 Subscription

Semantics

Subscription is the expression of interest in receiving and capability to receive a PubSubNotice.

Fully Scoped Name

EDOC::CCA::Event:: Subscription

Owned By

Subscriber

Properties

subscriptionClause

Expression based on attributes of PubSubNotice, describing the instance subset of interest to this subscription.

domain

A domain of interest. Only PubSubNotices produced within this domain are of interest.

Related Elements

Subscriber

A Subscription is *heldBy* exactly one Subscriber.

EventNotice

A Subscription *subscribesTo* one or more EventNotices.

FlowPort

A Subscription *Inherits* from FlowPort as per Component Model.

Constraints

SubscriptionClause Expression is constrained to the values of the attributes of the associated EventNotice. If the subscription is for more than one event notice, the expression is constrained to attributes that are common to all the event notices of interest.

3.11.4.5 PubSubNotice

Semantics

A PubSubNotice is any data structure that is *announcedBy* a publication and/or *subscribedTo* by a subscription. Instances of PubSubNotice are communicated as DataFlows from publishers to subscribers based on the subscriptions.

Fully Scoped Name

EDOC::CCA::Event:: PubSubNotice

Owned By

None

Properties

None

Related Elements

Subscription

A PubSubNotice is *subscribedBy* one or more Subscriptions.

Publication

A PubSubNotice is *announcedBy* one or more Publications.

CompositeData

A PubSubNotice *Inherits* from CompositeData as per Entities model.

Constraints

None

3.11.5 Event Package

This is an overview of event aspect of event-driven business computing. The yellow (shaded) elements are directly part of the event package. Each of them will be described in detail below. The white elements belong to other views and provide the context for this view.

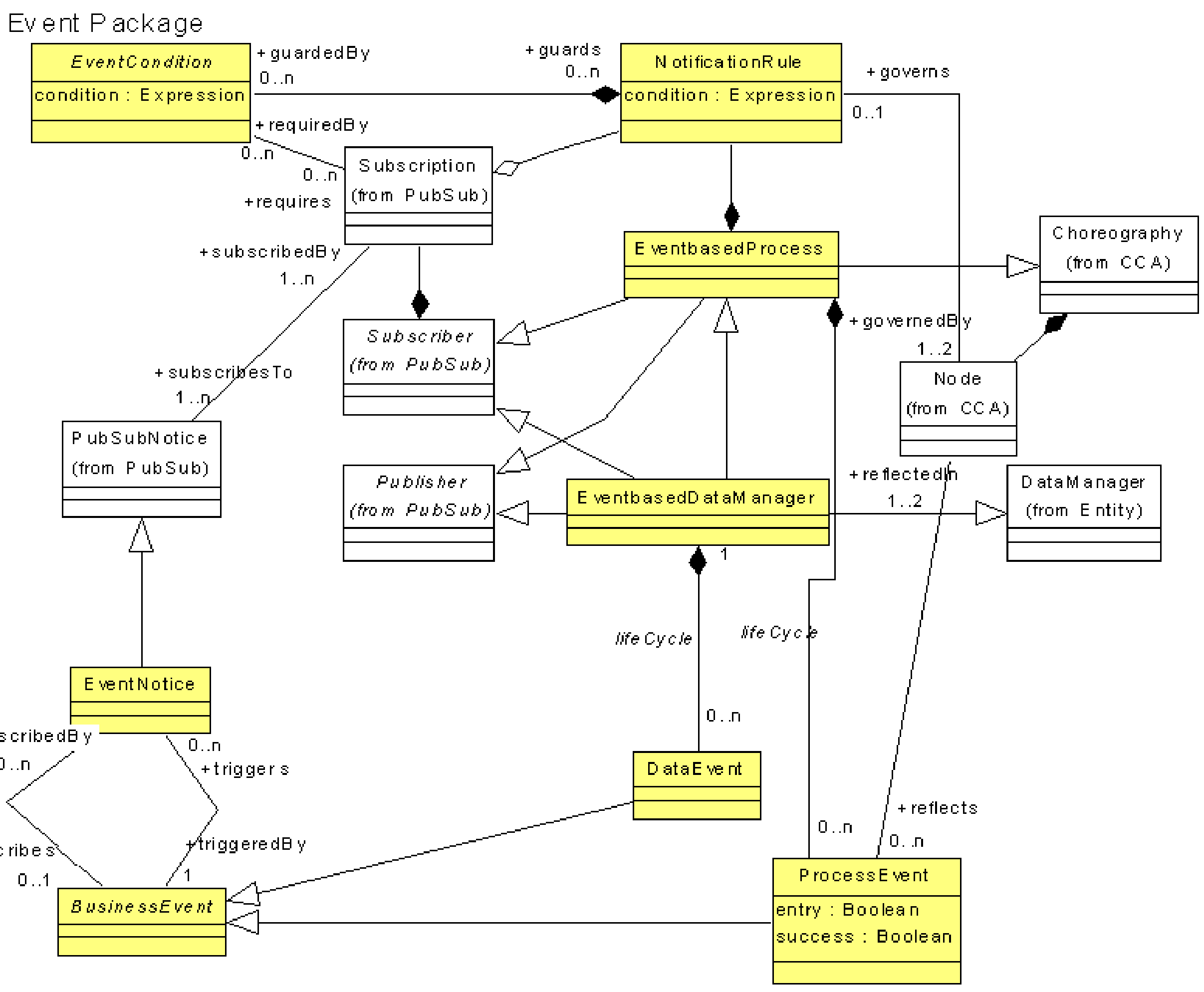


Figure 3-44 Diagram of Event Package

3.11.5.1 BusinessEvent

Semantics

A business event is any event of business interest that happens within an enterprise. BusinessEvents are either ProcessEvents or DataEvents.

Fully Scoped Name

EDOC::CCA::Event:: BusinessEvent

Owned By

None

Properties

None

Related Elements

EventNotice

A business event *triggers* zero or more event notices.

A business event is *describedBy* one or more event notices.

ProcessEvent

Business event is the Abstract supertype of ProcessEvent.

DataEvent

Business event is the Abstract supertype of DataEvent.

Constraints

None

3.11.5.2 ProcessEvent

Semantics

A process event is any business event that reflects a state change within a process (i.e., entry into or exit from Nodes in a Choreography).

Fully Scoped Name

EDOC::CCA::Event:: ProcessEvent

Owned By

EventBasedProcess

Properties

None

Related Elements

Node

A ProcessEvent reflects the entry into or exit from one Node (or the exit from one and entry into another, i.e., two Nodes).

BusinessEvent

ProcessEvent *Inherits* from BusinessEvent.

Constraints

Any Node referenced must belong to the EventBasedProcess that also owns this ProcessEvent.

3.11.5.3 DataEvent

Semantics

A data event is any business event that reflects a change in data managed by a DataManager.

Fully Scoped Name

EDOC::CCA::Event:: DataEvent

Owned By

EventBasedDataManager

Properties

None

Related Elements

BusinessEvent

ProcessEvent *Inherits* from BusinessEvent.

Constraints

None

3.11.5.4 EventNotice

Semantics

An event notice is any PubSubNotice that is triggered by a business event.

Fully Scoped Name

EDOC::CCA::Event:: EventNotice

Owned By

None

Properties

None

Related Elements

BusinessEvent

An event notice is *triggeredBy* exactly one Business Event.

An event notice may *describe* at most one Business Events.

PubSubNotice

An event notice *Inherits* from PubSubNotice.

Constraints

None

3.11.5.5 EventBasedProcess

Semantics

An EventBasedProcess is a subtype of Choreography (CCA model). It is a Subscriber and has NotificationRules associated with its Subscriptions. It is a Publisher and publishes ProcessEvents. ProcessEvents describe the life cycle of the EventBasedProcess.

Fully Scoped Name

EDOC::CCA::Event:: EventBasedProcess

Owned By

None

Properties

None

Related Elements

ProcessEvent

An EventBasedProcess owns a set of ProcessEvents that together describes the life cycle of the EventBasedProcess.

Choreography

An EventBasedProcess *Inherits* from Choreography (from CCA model).

Publisher

An EventBasedProcess *Inherits* from Publisher.

Subscriber

An EventBasedProcess *Inherits* from Subscriber.

EventBasedDataManager

An EventBasedProcess is the supertype of EventBasedDataManager.

Constraints

None

3.11.5.6 EventBasedDataManager

Semantics

An EventBasedDataManager is a DataManager. It is also a Publisher and publishes DataEvents when its data changes. It may also be a subscriber, typically subscribing to PubSubNotices relating to the maintenance of its data, e.g., replication.

Fully Scoped Name

EDOC::CCA::Event:: EventBasedDataManager

Owned By

None

Properties

None

Related Elements

DataEvent

An EventBasedDataManager owns a set of DataEvents that together describes possible changes to the data owned by the EventBasedDataManager.

DataManager

An EventBasedDataManager *Inherits* from DataManager (from Entities model).

Publisher

An EventBasedDataManager *Inherits* from Publisher.

Subscriber

An EventBasedDataManager *Inherits* from Subscriber.

EventBasedDataManager

An EventBasedDataManager *inherits* from EventBasedProcess.

Constraints

None

3.11.5.7 NotificationRule

Semantics

A NotificationRule is a rule associated with a subscription that determines what should happen within the EventBasedProcess holding the subscription when a qualifying PubSubNotice is delivered. Optionally, the NotificationRule can be further guarded by an EventCondition that requires the delivery of additional events.

Fully Scoped Name

EDOC::CCA::Event:: NotificationRule

Owned By

EventBasedProcess

Properties

Condition

An Expression based on attributes of PubSubNotice, describing the instance subset of the PubSubNotice that will cause the change in the EventBasedProcess indicated by this NotificationRule.

Related Elements

Subscription

A NotificationRule is associated with a Subscription and 'fires' upon receipt of the PubSubNotice associated with the Subscription.

EventCondition

A NotificationRule may be *guardedBy* one or more EventConditions calling for the receipt of additional events before this NotificationRule will 'fire' successfully.

The EventCondition guarding the NotificationRule must be satisfied before the NotificationRule's Condition is evaluated.

Node

A NotificationRule governs the entry into or exit from one Node (or the exit from one and entry into another, i.e., two Nodes).

Constraints

Any EventCondition must reference Subscriptions belonging to the same EventBasedProcess as the NotificationRule.

3.11.5.8 EventCondition

Semantics

An EventCondition identifies a subscription and specifies a PubSubNotice instance subset of which one must have been received to satisfy this condition.

Fully Scoped Name

EDOC::CCA::Event:: EventCondition

Owned By

NotificationRule

Properties

Condition

An Expression based on attributes of PubSubNotice, describing the instance subset of the PubSubNotice that will satisfy the guard constituted by this EventCondition.

Related Elements

Subscription

An EventCondition *requires* a Subscription and 'fires' upon receipt of a PubSubNotice associated with the Subscription. If the received PubSubNotice satisfies the condition expression, then the EventCondition has been satisfied.

Constraints

None

3.12 Relationship to other ECA Models

3.12.1 Relationship to Business Process Model and Entities Model

The ECA Business Process model describes a process as a set of activities.

Activities are defined in terms of responsible party, performers, artifacts, and pre and post conditions.

The Business Process model does not specify which performers act on what artifacts, and how.

It does not specify directly the relationship between states of artifacts and the pre and post conditions of activities.

It does not show directly what triggers each activity.

(Above three statements are qualified: other than as annotated in activity diagram as control flow and object flow.)

The Business Process model relies on components to implement the choreography. The states and transitions of choreography implement the control flows of the activity diagram.

The messages implement the information flows from the collaboration diagram.

The Events model (this model) describes events that happen to artifacts (entities). It describes business events as changes from one state to another. The Events model describes how activities result in state changes (i.e., events).

It describes how these BusinessEvents map to EventNotices, and how subscriptions can channel notifications to processes, and how delivery of an EventNotice can be mapped by NotificationRules to activities.

The Events model does not describe who or what within the process establishes the subscription, or who or what within the process reacts to receipts of notifications.

3.12.2 Relationship to ECA CCA Model

3.12.2.1 Modeling Events with Components

Events are changes in state to either entities or processes.

Just about anything that happens in a business has interest to someone else, and so every event (to an entity or to a process) has the potential for causing notification.

At the system level this means that any process or entity has to offer notification (i.e., allow subscription) to any of its state change notifications.

Most event notifications also trigger rules of some kind. If state of inventory changes to 'below-minimum-stock-level' some re-order rule kicks in. If state of the order-process changes to 'over-due' then some expediting rule kicks in.

At the system level this means that NotificationRules and BusinessConditions must be able to refer to events.

All activities result in a new state, or in failure.

At the system level this means that definitions of activities and operations include postconditions. These postconditions could be either expressions of events (i.e., state change) or more likely expression of state (where the state change, or event, is implicit).

The Events model relies on the CCA model to implement the outgoing event notification flows from an entity component, and the incoming event notification flows to a process component. Event notification flows happen from flow port to flow port.

The Events model relies on the CCA model to implement the linkage between (the completion of) an action on an entity and (an instance of) an event. The event model specifies which activity causes which event.

3.13 Relationship Other Paradigms

In general the central idea of event driven computing is that event notifications trigger action and/or communication, and that very little action or communication is not triggered by event notifications.

There are four main kinds of communication:

- Business notification: A one-way, information-only, notification. A special subtype is event-notification that informs that an event just happened. This is the main form of communication in event-driven computing.
- Query: A two-way, request, response, with the response being the query result set. This is a more tightly coupled model. However a query could be triggered by the loosely coupled receipt of a business notification. Also the gathering of data for a business notification could require one or more tightly coupled queries.
- Collaboration: A two-way, negotiation-style, communication that may or may not result in a new state between the parties. An atomic style subtype is the ebXML business transaction. This could be implemented in many ways. One way is to consider the requests and responses in the collaboration to simply be business notifications. Regardless how the collaboration itself is implemented, it could certainly be triggered by the loosely coupled receipt of a business notification. For instance notification of an event within the enterprise might trigger the collaboration to order more inventory.
- Method invocation: A one-way, with optional return parameters, communication that usually causes the state at the remote end to change in a predefined way. Again, a method invocation could be triggered by the loosely coupled receipt of a business notification. Also under event driven computing entity operations, which are often implemented as method invocations, will trigger the sending of one or more loosely coupled event notifications. A cousin of method invocation, web service invocation, is usually likely to be implemented as one way transfer of messages over standard internet protocols. As such you could easily have web services react directly to event notifications.

So again, event notifications can trigger many kinds of communication, and based on business rules and or subscriptions, the kind of communication may be another notification, a collaboration, a method invocation, or a query.

Many times a tightly coupled systems model can be replaced with an event based model to create more flexibility in business and systems re-engineering. Generically replacing state machines with event-driven computing always adds loose coupling. In a state machine, the event is both the thing that happened and the stimulus for something else to happen. The two cannot be separated. In event driven computing the event, the sending of a notification, the receipt of the notification, and the reaction to the notification are all separate, and can be much more easily reconfigured upon demand.

The above is true both at a generic business level and at a system level.

3.13.1 ebXML

ebXML is a large initiative to model and implement business collaborations based on XML message exchanges between the parties.

There are several relationships of the event model to ebXML.

First, event driven computing within the enterprise is the best way to determine when to initiate business collaborations.

Second, the XML message exchanges could themselves be treated as business notifications.

Thirdly, the ebXML business model is based in part on a model for exchange of economic resources, where each such exchange is called an economic event. The capture of such economic events is similar to the capture of normal business events, and the communication of the notifications can be the same for both.

Fourth, the model for economic resources deals also with future commitments, which can be thought of as promises to execute economic events in the future. This extends the event model into prediction of events and executions against those predictions.

ebXML, phase one, was approved in May of 2001. In this phase, the ebXML business process choreography is already near identical to the ECA choreography. It is predicted that ebXML phase two will bring further alignment to ECA, and to the evolving web services standards.

3.14 Example

In the engineering of EventBasedProcesses you identify the business entities to be affected and examine their available business events and ‘communicated’ business notifications. Activities for the EventBasedProcess are then constructed to contain NotificationRules that ‘listen’ for the appropriate business notifications, and business activities that cause the appropriate business events to happen. The process can easily be re-engineered by changing the subscriptions, or the NotificationRules, thus causing different business activities to happen in response to a given business notification.

A basic EventBasedProcess, and its relationships to business entities can be depicted on a diagram such as that below.

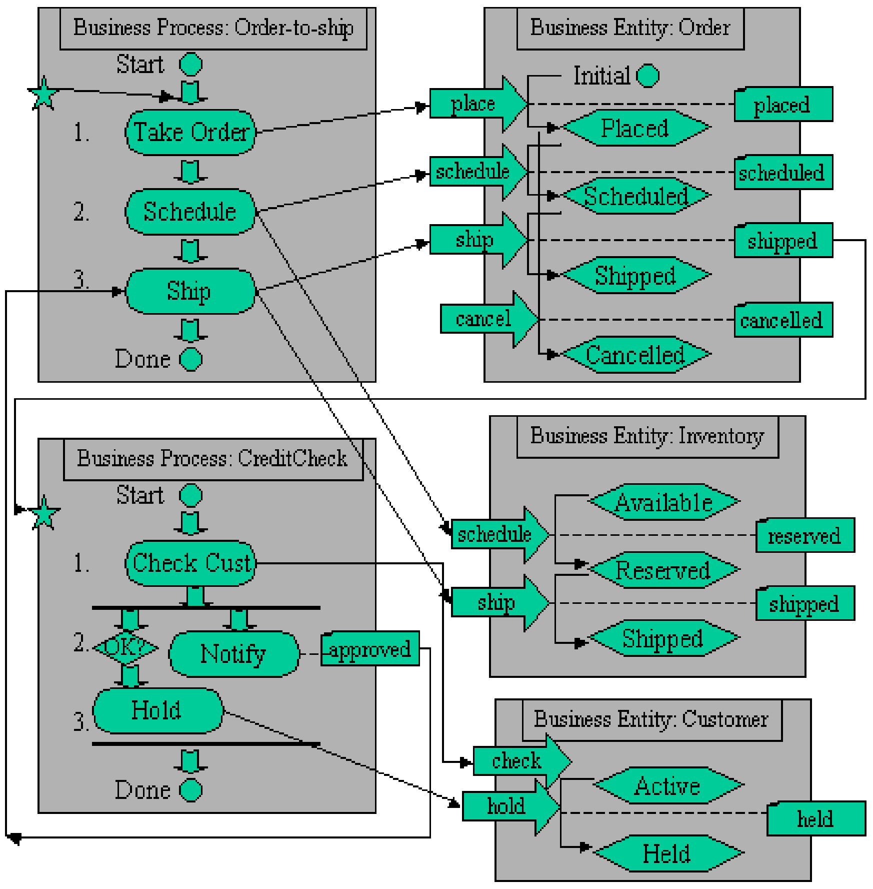


Figure 3-45 Business process/entity/event diagram

Processes and entities are depicted as large boxes. Activities within a process are ovals. Events are 'dog-eared' boxes. Entity operations are fat arrows. Entity states are hexagons. Business notifications are arrows from event boxes to the left side of process boxes. Invocations of entity operations are arrows from activity ovals to the fat arrows.

This diagram contains notational elements that can (almost) all be mapped directly to an Activity Diagram for the EventBasedProcess, a State Chart for the Entity, and a Sequence Diagram for the interaction between the two.

Section V - The Business Process Model

The Business Process model specializes the CCA, and describes a set of UML extensions that may be used on their own, or in combination with the other EDOC elements, to model system behavior in the context of the business it supports.

3.15 Introduction

The Business Process model provides modeling concepts that allow the description of business processes in terms of a composition of business activities, selection criteria for the entities that carry out these activities, and their communication and coordination. In particular, the Business Process model provides the ability to express:

- Complex dependencies between individual business tasks (i.e., logical units of work) constituting a business process, as well as rich concurrency semantics.
- Representation of several business tasks at one level of abstraction as a single business task at a higher level of abstraction and precisely defining relationships between such tasks, covering activation and termination semantics for these tasks.
- Representation of iteration in business tasks.
- Various time expressions, such as duration of a task and support for expression of deadlines.
- Support for the detection of unexpected occurrences while performing business tasks that need to be acted upon (i.e., exceptional situations).
- Associations between the specifications of business tasks and business roles that perform these tasks and also those roles that are needed for task execution.
- Initiation of specific tasks in response to the occurrence of business events.
- The exposure of actions that take place during a business process as business events.

3.16 Metamodel

This model is organized with three main model elements to describe a business process: **BusinessProcess**, **CompoundTask,** and **Activity** as shown in Figure 3-46 in which the derivation from the CCA is shown. BusinessProcess is the outermost layer of composition representing a complete process specification. It is a ProcessComponent for the purpose of its usage inside other CCA Compositions, but its Composition is constrained in the same way as a CompoundTask. In other words, BusinessProcesses are the entry point from CCA to a process definition. CompoundTasks are also specializations of CCA ProcessComponents, but their Ports are constrained specializations of CCA Ports that represent the data required to initiate an enactment of its Composition, which defines how it executes.

The only ComponentUsages CompoundTasks and BusinessProcesses may contain are Activities, which are specializations of CCA ComponentUsages. Activities are the pieces of work required to complete a Process, and CompoundTasks are the containers for a logical set of Activities and the **DataFlows** that define the temporal and data dependencies between them. DataFlows are specializations of CCA Flows that connect the PortConnectors on the Activities.

Activities are always usages of a CompoundTask definition, which defines the Port types and their correlation semantics. CompoundTasks defining an Activity either compose additional Activities and DataFlows to show how this Activity is performed, or the Activity also refers to a **Performer ProcessRole** via the **performedBy** association, which is a binding to a ProcessComponent that fulfills the requirements of the **ProcessRole**. Performer ProcessRoles are the exit point from a process definition that allows it to invoke ProcessComponents (and their specializations, such as Entities). Many Activities may be usages of the same CompoundTask definition, and many activities in the same CompoundTask may be performed by the same **ProcessRole**.

(See Section 3.19 for the combined Process Model)

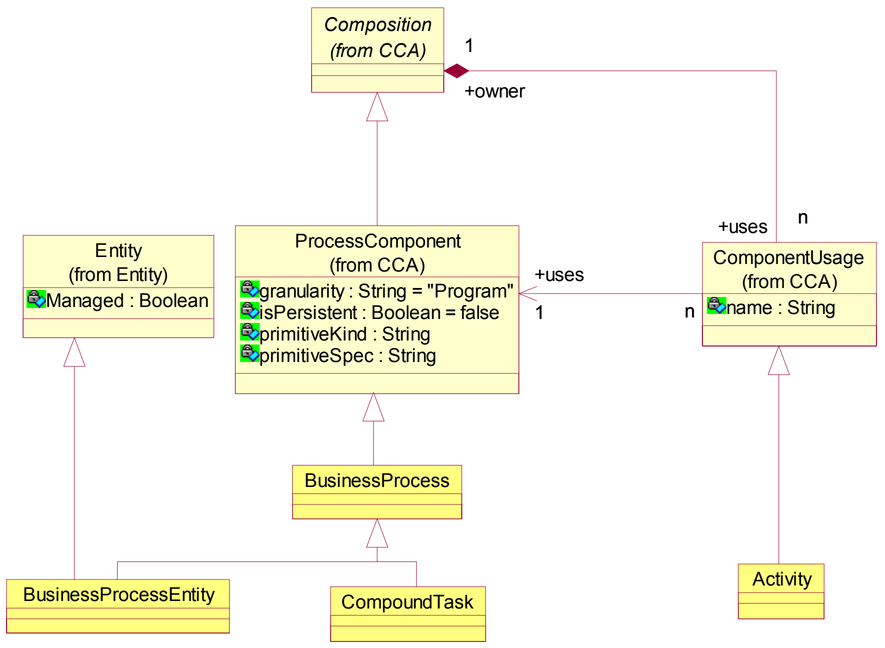


Figure 3-46 Composition of Process ModelElements

DataFlows (constrained Flows) allow the connection of the ProcessPortConnectors representing the ProcessFlowPorts of a CompoundTask to the ProcessPortConnectors of its contained Activities and vice versa. We will call the ProcessPortConnectors representing usage of a ProcessFlowPort in an InputGroup **input ProcessPortConnectors**. Likewise the ProcessPortConnectors representing usage of a ProcessFlowPort in an OutputGroup or ExceptionGroup are called **output** and **exception ProcessPortConnectors**, respectively.

The flow of data typically goes from input ProcessPortConnectors of the CompoundTask to the input ProcessPortConnectors of an Activity contained by the CompoundTask, and then from the output ProcessPortConnectors of the Activity to either the input ProcessPortConnectors of another contained Activity or to the output or exception ProcessPortConnectors of the CompoundTask.

ProcessFlowPorts are the formal types of inputs to and outputs from a CompoundTask. They have a multiplicity, given by the attribute pair **multiplicity_lb** and **multiplicity_ub**, which indicates the lower bound on the number of values that needs to be received or transmitted by the PortConnector instantiating this port type at runtime, as well as the upper bound on the number of values that the PortConnector can hold before it begins discarding them.

Multiports are used to aggregate FlowPorts. The MultiPort specializations, InputGroup, OutputGroup, and ExceptionGroup indicate that a set of ProcessFlowPorts, when used in some Composition, must all receive values from DataFlows before any of the values are received or transmitted by the CompoundTask that owns them. They can be considered to be correlators. A ProcessMultiPort may be synchronous or asynchronous, as indicated by its **synchronous** attribute inherited from Port. Usages of Synchronous ProcessMultiPorts indicate the initiation or termination of the execution of some Activity owning the PortUsage, whereas usages of asynchronous ProcessMultiPorts may only have the values in their contained ProcessFlowPorts transmitted into or out of an already executing Activity.

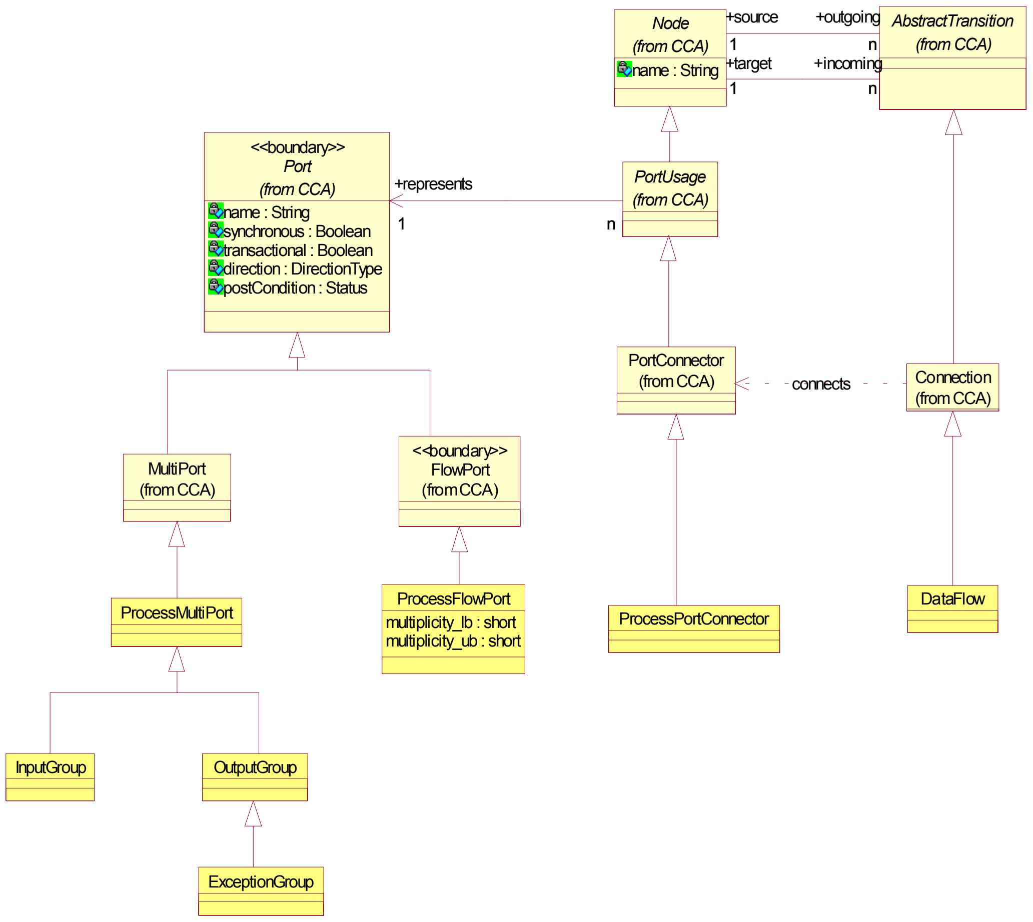


Figure 3-47 Inputs and Outputs of Process ModelElements

In addition:

- An Activity may specify required Artifact(s) that select information entities to be used or produced.
- An Activity may specify ResponsibleParty(s) that select people, company, or other group roles that are responsible for the Activity.
- Each Activity may have ActivityPreCondition(s) and ActivityPostCondition(s) that further constrain when it starts and how it completes (see Section 3.18, “Process Model Patterns,” on page 3-152).

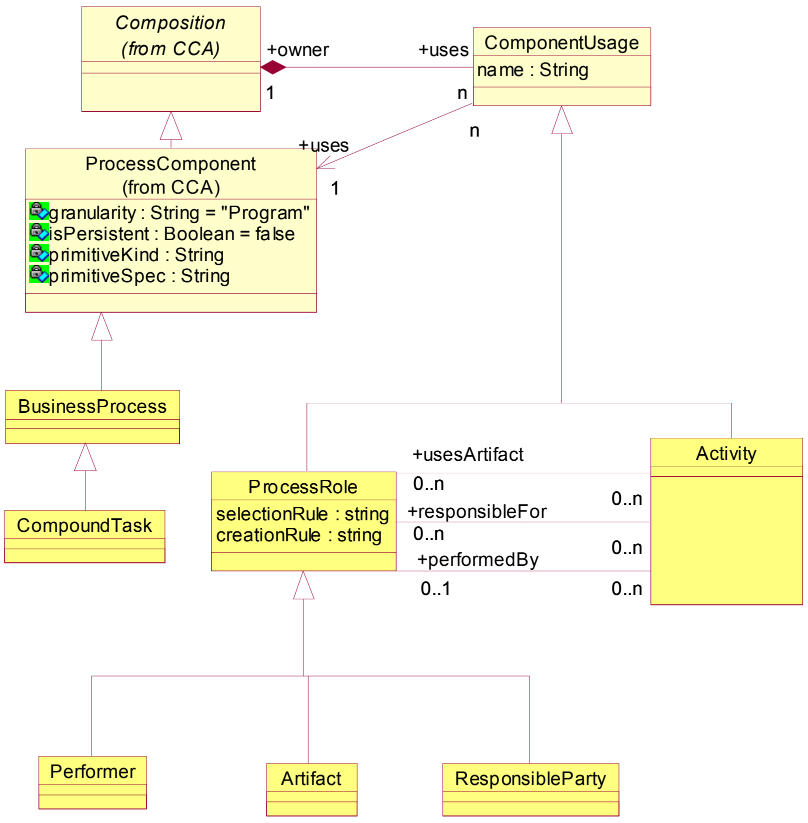


Figure 3-48 Diagram of the Roles aspect of the Process Model

The model in Figure 3-48 shows the ownership of **ProcessRoles** by CompoundTasks (via their ProcessComponent base class). ProcessRoles have three kinds of relationships with Activities. An Activity may be **performedBy** a ProcessRole, or it is possible that an Activity has a **usesArtifact** association with a ProcessRole, or a ProcessRole may be responsible for an Activity, as indicated by a **responsibleFor** association end role. The same ProcessRole may have several associations with different Activities, for example to be the performer for one activity, while also being an artifact for another, or to be both the responsible party and performer for an Activity. The specific ProcessRoles of Performer, Artifact, and ResponsibleParty are constrained to be associated with Activities only by the performedBy, usesArtifact, and responsibleFor associations respectively, and are useful in many cases where ProcessRoles do not need to be re-used.

At run time a ProcessRole represents the binding of a state variable in its owner CompoundTask to a concrete ProcessComponent instance that meets the requirements of the **selectionRule** or **creationRule** attributes of the ProcessRole. Typically the performer roles of an Activity will have a type from which the defining CompoundTask of that Activity has been derived. The OperationPorts of the ProcessComponent identified by the ProcessRole will be represented as a pair of an

InputGroup and an OutputGroup that contain ProcessFlowPorts that represent the input and output parameters of the OperationPort. Exceptions are represented by additional ExceptionGroups.

In addition to the basic set of model elements given above, there are a number of other important concepts required in the modeling of Processes that can be expressed as patterns of use of these basic elements:

- ActivityPreCondition
- ActivityPostCondition
- Timeout
- Terminate
- Loops
  - Simple Loop
  - While and Repeat/Until Loop
  - For Loop
- Multitask

These are explained in Section 3.18, “Process Model Patterns,” on page 3-152.

An example of a CompoundTask containing Activities is shown in Figure 3-49.

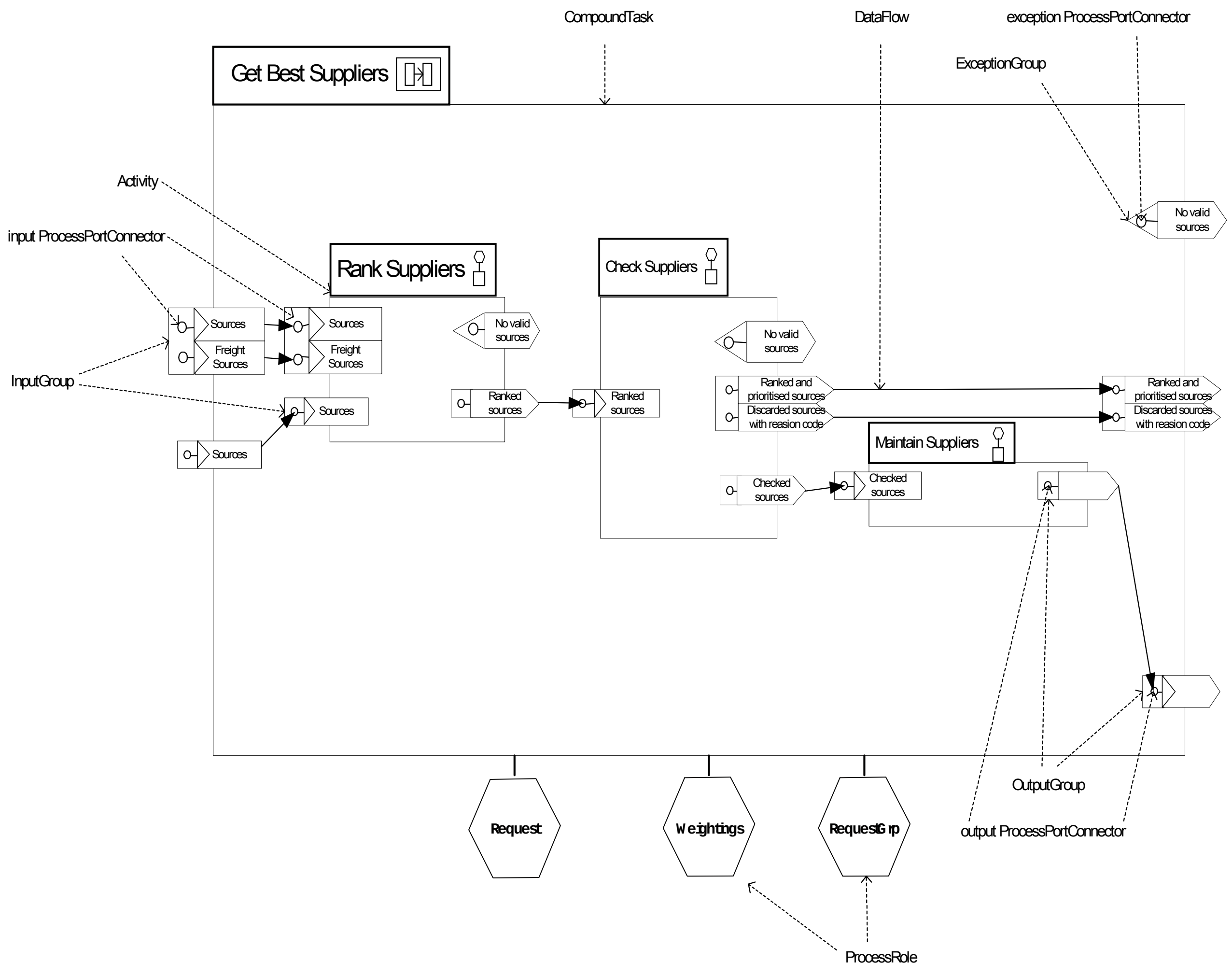


Figure 3-49 A labeled CompoundTask Diagram

3.16.1 Business Process Metamodel

The metamodel for the Business Process model is contained in a single package, BusinessProcess.

3.16.1.1 CompoundTask

Semantics

A CompoundTask defines how to coordinate a set of related Activities that, in combination, perform some larger scale activity, ultimately in the context of a Business Process. It represents the formal type and Correlation Protocol Contract of Ports available on Activities that use the CompoundTask. It is also a container

(Composition) of Activities that use other CompoundTasks (or, when describing recursion, that re-use this CompoundTask), a container of the DataFlows between these Activities, and the ProcessRoles that model bindings to Objects required by these Activities.

Fully Scoped name

ECA::BusinessProcess::CompoundTask

Owned by

Inheritance

ECA::BusinessProcess::BusinessProcess

  CompoundTask

Properties

Associated elements

Constraints

[1] All Ports owned by a CompoundTask must be ProcessMultiPorts.

[2] All ComponentUsages contained by a CompoundTask must be Activities or ProcessRoles.

3.16.1.2 Activity

Semantics

Activity represents the execution of a part of a Business Process using one of two mechanisms (but not both). The mechanisms are:

- The creation of a Composition of nested Activities, ProcessRoles, and DataFlows described by the CompoundTask that the Activity references through its uses association.
- The execution of some feature of an Object bound to a ProcessRole instance referred to via the Activity's **performedBy** association. (See Section 3.16.1.12, "ProcessRole," on page 3-147.)

Hence an Activity represents an action that is either described by a further decomposition in the form of a CompoundTask or it represents and action that is performed by objects bound to ProcessRoles either statically, or at runtime as the Activity enters the Running state.

An Activity may also be associated via the usesArtifact and responsibleFor associations to one or more ProcessRoles. These ProcessRoles will be bound to Objects at run time as the Activity enters the Running state.

An Activity's PortUsages representing InputGroups (input PortUsages), which contain ProcessPortConnectors representing ProcessFlowPorts (input ProcessPortConnectors), are the alternative means by which the Activity may supply data to these mechanisms to initiate some action.

PortUsages representing synchronous InputGroups owned by an Activity instance represent different initializations, and only one of these will ever be enabled, at which time the Activity instance will begin its execution.

An Activity instance must be in the Running state before it can use any data in input PortUsages (synchronous or asynchronous) from its containing Activity instance.

If no Synchronous input PortUsages are present, then the Activity will be initialized as part of the initialization of its container Activity. This will allow it to receive asynchronous inputs as soon as they propagate into the container Activity.

When an Activity is performedBy a ProcessRole that has not yet been bound, the ProcessRole will be bound to an appropriate Object during the initialization of the Activity. The binding for the Role will last at least for the duration of the life time of the Activity, but the Object it binds to may exist before the binding is created, and may live longer than the binding. Once bound, the Role will persist until all other Activities to which it is associated have completed.

Asynchronous input PortUsages owned by an Activity represent the means by which the Activity may accept input values during its active life time. When an Activity is in the NotStarted state (none of its synchronous input PortUsages is enabled) all data values that arrive at a ProcessPortConnector in an asynchronous PortUsage will be kept in that Port Connector only up to its multiplicity's upper bound. Additional values will cause discarding. However, once the Activity enters the Running state the sets of correlated Inputs will be consumed by the Activity.

Note – This behavior trades off the resource savings of keeping asynchronous values only up to and including the slots defined by an Input's multiplicities against the ability to queue all asynchronous flows on behalf of Activities yet to be enabled. The problem is that in many process definitions, choices are made about which path a process will take, leaving many Activities' input PortUsages only partially satisfied and unable to ever become enabled. In a long-lived Process this may mean that large numbers of data values arriving at asynchronous Inputs will be queued, never to be consumed by that Activity.

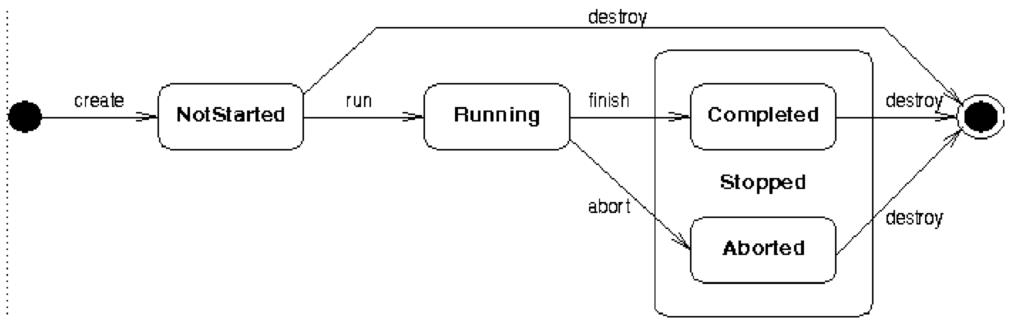


Figure 3-50 State Machine describing execution of Activities and CompoundTasks

Runtime Semantics: Figure 3-50 shows the state machine for an Activity instance. When an Activity is created, only the resources required to enact the PortUsage behavior of the Activity are created. The Activity then enters the NotStarted state. In this state the Activity may accept Flows at its input ProcessPortConnectors.

Once one of its synchronous input PortUsages is enabled (or it has no synchronous input PortUsages), ProcessRole binding is performed (as specified for ProcessRole in Section 3.16.1.12, “ProcessRole,” on page 3-147).

Then, if the Activity uses a CompoundTask that is a non-empty Composition, all the resources to represent the contained DataFlows, Bindings, and nested Activities are allocated and all nested Activities are created. The Activity now enters the Running state.

An Activity instance enters the Completed state when none of its contained Activity instances that have synchronous output PortUsages containing values (that are not also exception PortUsages) are in the Running state and there are no DataFlows that are in the process of delivering their data (which could then trigger the running of another Activity). Note, this means that not all contained Activities need to have executed, only that none (that have synchronous output ProcessMultiPorts) are running. This results in a **quiescent** model for completion.

Alternatively, if an Activity instance has an exception PortUsage that is satisfied, then all Activity instances that are contained by this Activity instance and are in the Running state are aborted. The Activity will then satisfy the quiescent model completion criteria just outlined.

An Activity instance enters the Completed state, if a satisfied synchronous output PortUsage is enabled. If there is more than one satisfied synchronous output PortUsage, then the choice of which one to enable is arbitrary. If there is no synchronous output PortUsage that is satisfied, then the Activity instance’s system ExceptionGroup is enabled.

If a nested Activity instance contained by an Activity enters the Completed state with an exception PortUsage enabled and the exception is unhandled (see Section 3.16.1.11, “ExceptionGroup,” on page 3-146 for the definition of handled and unhandled ExceptionGroups), then the containing Activity instance’s system ExceptionGroup is enabled.

If an Activity instance is aborted, it terminates all of its contained Activity instances and enters the Aborted state.

If the Activity uses an empty Composition, it must have a performedBy link to a ProcessRole, which will now be bound, and the Activity instance enters the Running state. While in the Running state, values from enabled input ProcessPortConnector instances may be consumed. In most cases this will mean that the PortUsage that was enabled has a collection of input Parameters for a method on the Object bound to the performer ProcessRole, which will be invoked. The return of the method will place values into an output PortUsage (representing an OutputGroup or ExceptionGroup), which will enable that PortUsage.

The Activity instance enters the Stopped state when one of its synchronous OutputGroup instances is enabled. If this is an ExceptionGroup instance, then it enters the Aborted state, otherwise it enters the Completed state.

Fully Scoped name

ECA::BusinessProcess::Activity

Owned by

CompoundTask

Inheritance

ECA::CCA:: ComponentUsage

  Activity

Properties

Associated elements

uses (from ComponentUsage)

An Activity is always associated with a CompoundTask via the uses association.

performedBy

An Activity with an empty CompoundTask Composition must be linked to a single ProcessRole via the performedBy association.

usesArtifact

An Activity may require access to Objects via a ProcessRole to use as a passive resource. Its usesArtifact association indicates the Roles it uses for this purpose.

responsibleFor

An Activity in a BusinessProcess may be performedBy a ProcessRole that does so on behalf of another Role or Roles that are responsible for the Activity. The responsibleFor association allows these Roles to identify Object representing responsible parties.

Constraints

[1] An Activity that uses a CompoundTask definition with no internal Composition must have a performedBy link.

3.16.1.3 BusinessProcess

Semantics

A BusinessProcess defines the ProcessComponent view of a process definition that coordinates a set of related Activities. It defines a complete business process that can be invoked from another CCA Composition, usually using OperationPorts, which are connected via DataFlows (a subtype of CCA Flow) to the ProcessPortConnectors of the Activities that it contains. In other words, a BusinessProcess is an ordinary ProcessComponent on the outside and a CompoundTask on the inside.

Fully Scoped name

ECA::BusinessProcess::BusinessProcess

Owned by

Inheritance

ECA::CCA::ComponentDefinition::ProcessComponent

  BusinessProcess

Properties

Associated elements

Constraints

All ComponentUsages contained by a BusinessProcess must be Activities.

All Connectors contained by a BusinessProcess must be DataFlows.

3.16.1.4 BusinessProcessEntity

Semantics

A BusinessProcessEntity is a BusinessProcess that is also an Entity with identity. It is used to model long-lived processes that may require management and or interaction during their lifetime.

Fully Scoped name

ECA::BusinessProcess::BusinessProcessEntity

Owned by

Inheritance

ECA::BusinessProcess::BusinessProcess

  BusinessProcessEntity

ECA::Entity::Entity

  BusinessProcessEntity

Properties

Associated elements

Constraints

N/A

3.16.1.5 ProcessFlowPort

Semantics

ProcessFlowPort represents data used in CompoundTask input/output.

Runtime Semantics: A ProcessFlowPort instance (represented by a ProcessPortConnector on an Activity) is **satisfied** when it has at least multiplicity_lb values, otherwise it is **unsatisfied**. It may not have more than multiplicity_ub values.

If a ProcessFlowPort instance is the sink of more than one DataFlow, then data values for that instance can be supplied by any one of those DataFlows up to the upper bound its multiplicity. In the default case of a multiplicity of {1,1} this implies OR semantics. If more values are supplied than the multiplicity's upper bound, the ProcessFlowPort instance's collection remains at the size of the upper bound, and some arbitrary set of values are discarded.

When a ProcessFlowPort instance is enabled and its containing CompoundTask instance (represented by an Activity) is in the Running state, it transmits its values using all the associated DataFlows (*AND semantics*) as appropriate. If the

ProcessFlowPort instance is contained by an asynchronous InputGroup instance, it then discards its values and resets its state to unsatisfied or satisfied according to its multiplicity.

Fully Scoped name

ECA::BusinessProcess::ProcessFlowPort

Owned by

ProcessMultiPort

Inheritance

ECA::CCA::FlowPort

  ProcessFlowPort

Properties

multiplicity_lb : short

multiplicity_ub : short

The multiplicity of a ProcessFlowPort instance allows it to act as a collection of data values of the same type. A multiplicity is expressed as a lower-bound, upper-bound pair {multiplicity_lb, multiplicity_ub}, where -1 is used in the upper bound to indicate infinity. The default multiplicity is {1,1} which represents a singleton collection.

Associated elements

ECA::CCA::DocumentModel::DataElement

A ProcessFlowPort is optionally associated with a DataElement by the **type** association, which is inherited from FlowPort. A ProcessFlowPort that does not have an associated type can be thought of as a **control point**. That is, the values handled by these ProcessFlowPorts are like objects that have identity but no attributes. They can be used, in conjunction with DataFlows, to describe control flow constraints that do not involve data values.

Constraints

[1] A ProcessFlowPort must be owned by a ProcessMultiPort.

3.16.1.6 ProcessPortConnector

Semantics

A ProcessPortConnector represents the usage of a ProcessFlowPort in the context of a CompoundTask.

Fully Scoped name

ECA::BusinessProcess::ProcessPortConnector

Owned by

CompoundTask

Inheritance

ECA::CCA::ComponentDefinition::PortConnector

ProcessPortConnector

Properties

Associated elements

represents (from PortUsage)

A ProcessPortConnector is always associated with a ProcessFlowPort via the represents association.

outgoing (from Node)

A ProcessPortConnector may be associated with zero or more DataFlows via the outgoing association.

incoming (from Node)

A ProcessPortConnector may be associated with zero or more DataFlows via the incoming association.

Constraints

[1] All Ports associated with a ProcessPortConnector by the represents association must be ProcessFlowPorts.

[2] All ProcessPortConnectors must be owned by CompoundTasks.

3.16.1.7 DataFlow

Semantics

A DataFlow represents a causal relationship in a business process. The source of the DataFlow must “happen” before the sink of the DataFlow. DataFlows also propagate data values between causally related ProcessPortConnectors. In the case that a DataFlow connects two ProcessPortConnectors in synchronous ProcessMultiPorts, the implication is that the Activities occur in strict temporal sequence.

Runtime Semantics: A DataFlow instance is created when its containing CompoundTask instance is created.

The enabling of the source of a DataFlow causes the enabling of the DataFlow, which then propagates the values from the source ProcessPortConnector to the sink ProcessPortConnector. The sink ProcessPortConnector may then discard values as necessary if its multiplicity upper bound is reached.

Fully Scoped name

ECA::BusinessProcess::DataFlow

Owned by

CompoundTask

Inheritance

ECA::CCA:: Connection

DataFlow

Properties

Associated elements

Constraints

[1] A ProcessPortConnector is a source of a DataFlow. A DataFlow has exactly one source ProcessPortConnector, but a ProcessPortConnector can be the source of zero or more DataFlows.

[2] A ProcessPortConnector is a sink of a DataFlow. A ProcessPortConnector has exactly one sink ProcessPortConnector, but a ProcessPortConnector can be the sink of zero or more DataFlows.

[3] The ProcessPortConnector that is the source of a DataFlow must be contained (indirectly) by the same CompoundTask as the DataFlow, and must be either:

- a ProcessPortConnector representing a ProcessFlowPort of an InputGroup of the CompoundTask; or
- a ProcessPortConnector representing a ProcessFlowPort owned by a PortUsage representing an OutputGroup of a CompoundTask used by an Activity directly contained by the DataFlow's containing CompoundTask.

[4] A ProcessPortConnector that is the sink of a DataFlow must be contained (indirectly) by the same CompoundTask as the DataFlow, and must be either:

- a ProcessPortConnector representing a ProcessFlowPort of an OutputGroup of the CompoundTask; or
- a ProcessPortConnector representing a ProcessFlowPort owned by a PortUsage representing an InputGroup of a CompoundTask used by an Activity directly contained by the DataFlow's containing CompoundTask.

The well-formedness rules above can be considered as reading "DataFlows cannot cross the boundaries of CompoundTasks." Figure 3-51 shows three illegal DataFlows (Note how the illegal DataFlows cross Task boundaries).

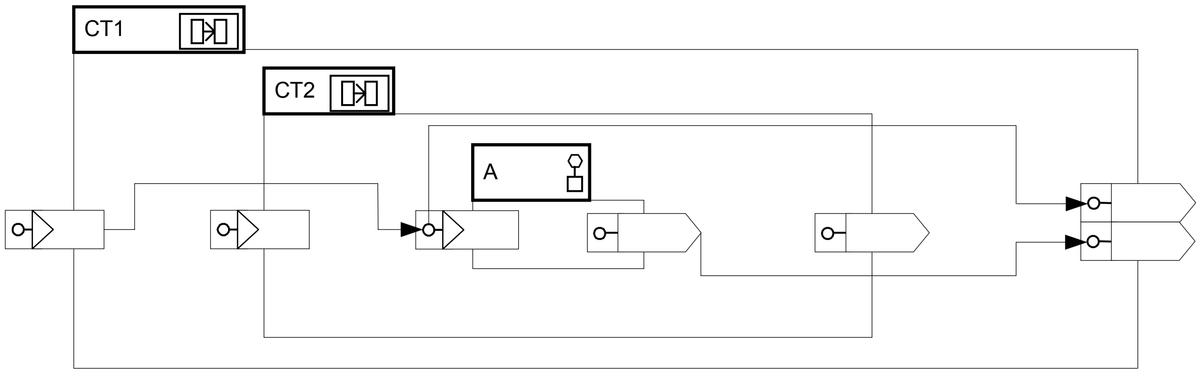


Figure 3-51 Illegal DataFlows crossing Task boundaries

[5] The type of the ProcessFlowPort represented by the source ProcessPortConnector of a DataFlow must be the same as (or coerce-able to) the type of the ProcessFlowPort represented by the sink ProcessPortConnector of a DataFlow. Coercible includes converting a value of type T to a member of type collection<T>
and vice versa.

[6] DataFlows between ProcessPortConnectors owned by PortUsage representing synchronous ProcessMultiPorts within a CompoundTask should be acyclic; that is, things cannot happen in a circular order. (However, see Business Process Patterns in Section 3.18, "Process Model Patterns," on page 3-152 for how to specify processes involving looping.)

3.16.1.8 ProcessMultiPort

Semantics

ProcessMultiPort represents a set of related ProcessFlowPorts used to describe the inputs and outputs of CompoundTasks. They act as a form of correlator for DataFlows.

Run-Time Semantics: As this section describes the semantics of ProcessMultiPorts, owned by CompoundTasks, we use the terminology **ProcessMultiPort instance** to mean a PortUsage representing a ProcessMultiport owned by an Activity, which we call a **CompoundTask instance**. In the same way the term **ProcessFlowPort instance** is used to mean a ProcessPortConnector contained by the PortUsage representing the ProcessMultiport.

A ProcessMultiPort instance is satisfied when **all** of its contained ProcessFlowPort instances are satisfied (AND semantics), otherwise it is unsatisfied.

If a ProcessMultiPort instance is satisfied, then it may be **enabled**. However, at most one synchronous InputGroup instance of a CompoundTask instance and one synchronous OutputGroup instance of a CompoundTask instance may be enabled and, once enabled, must remain in that state. An asynchronous ProcessMultiPort instance does not have these constraints. It will enable its ProcessFlowPort instances whenever it becomes enabled allowing them to transfer their contents and reset their state to unsatisfied (or satisfied if their multiplicity_lb is zero). This semantics is described formally using the Protocol in Figure 3-52 .

See the definitions of InputGroup in Section 3.16.1.9, “InputGroup,” on page 3-144, and OutputGroup in Section 3.16.1.10, “OutputGroup,” on page 3-145 for more specific behavioral specifications.

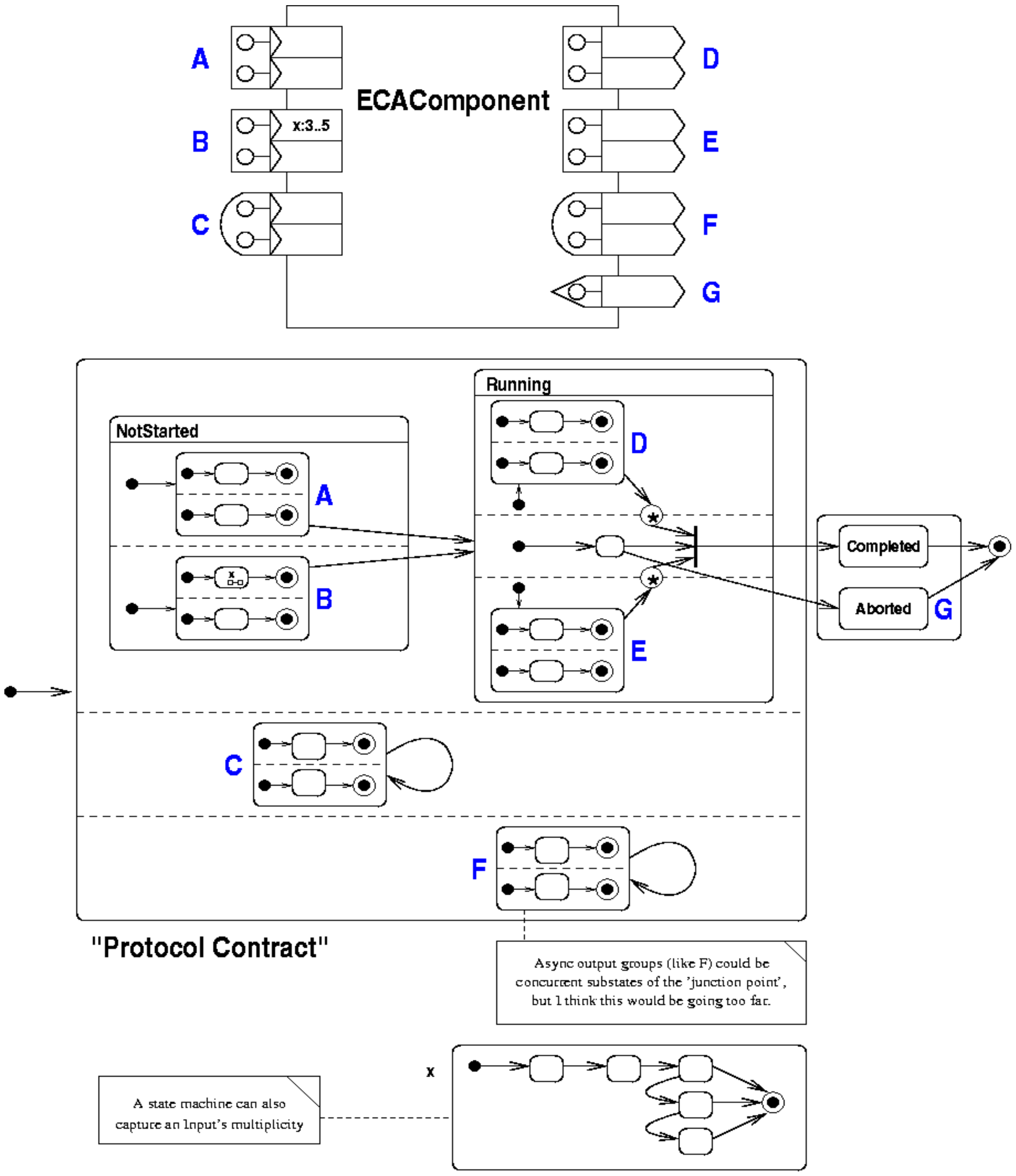

ECAComponent
A
B
x:3..5
C
D
E
F
G
NotStarted
Running
Completed
Aborted
"Protocol Contract"
Async output groups (like F) could be concurrent substates of the 'junction point', but I think this would be going too far.
A state machine can also capture an Input's multiplicity
x

Figure 3-52 Example Protocol describing the behavior of ProcessMultiPorts

Fully Scoped name

ECA::BusinessProcess::ProcessMultiPort

Owned by

CompoundTask

Inheritance

ECA::CCA::MultiPort

ProcessMultiPort

Properties

synchronous : boolean (from Port)

A value of TRUE indicates that this ProcessMultiPort represents either parameters that may be used to trigger a CompoundTask instance to enter the Running state, or results that are available when the instance enters the Stopped state.

A value of FALSE indicates that while the CompoundTask instance is in the Running state, the ProcessMultiPort may either asynchronously consume one or more sets of data, or asynchronously emit one or more sets of data.

Associated elements

ProcessFlowPort

A ProcessMultiPort provides a correlation framework for a number of ProcessFlowPorts.

Constraints

[1] The Composition owning a ProcessMultiPort must be a CompoundTask.

3.16.1.9 InputGroup

Semantics

InputGroup is a specialization of ProcessMultiPort. It is a container for a number of ProcessFlowPorts, which are the inputs to a CompoundTask.

Runtime Semantics: The InputGroup implies special semantics for the lifecycle of an Activity using the CompoundTask definition that owns it when its **synchronous** attribute is TRUE. In this case the InputGroup must be enabled before the Activity may enter its Running state.

Fully Scoped name

ECA::BusinessProcess::InputGroup

Owned by

CompoundTask (via its base class Composition)

Inheritance

ECA::CCA::ComponentSpecification::ProcessMultiPort

InputGroup

Properties

Associated elements

Constraints

3.16.1.10 OutputGroup

Semantics

OutputGroup represents a possible outcome of a CompoundTask; it provides data values associated with that outcome. In the case of a synchronous OutputGroup it also serves as an indication that an Activity using the CompoundTask definition to which the OutputGroup belongs has entered the Stopped state.

OutputGroup models a collection of data values produced by a CompoundTask.

Runtime Semantics: The OutputGroup implies special semantics for the lifecycle of an Activity using the CompoundTask definition that owns it when its **synchronous** attribute is TRUE. In this case the Activity must be in its Stopped state before the OutputGroup may be enabled.

Fully Scoped name

ECA::BusinessProcess::OutputGroup

Owned by

CompoundTask

Properties

Associated element

Constraints

3.16.1.11 ExceptionGroup

Semantics

ExceptionGroup represents the outcome of a CompoundTask that failed to complete its function. In a CompoundTask, an Activity's ProcessPortConnectors representing the ProcessFlowPorts of ExceptionGroup can be handled either by an exception handler (an Activity) to which the Port Connectors have DataFlows, or by an ExceptionGroup of the containing CompoundTask to which it has DataFlows. If, at runtime, an Activity's ExceptionGroup is not handled and the Exception is enabled, then it will be **propagated**. That is, the containing CompoundTask instance's system Exception will be enabled (which consequently causes the CompoundTask instance to abort its contained Activities and terminate in the Aborted state).

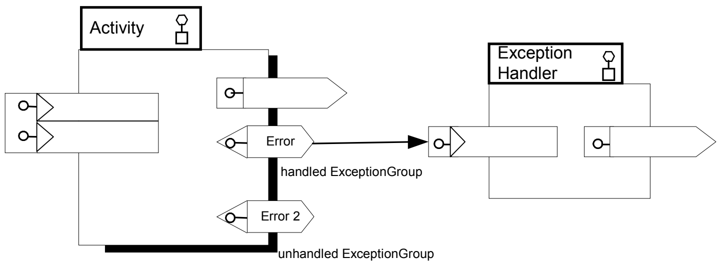


Figure 3-53 An ExceptionGroup that is handled by and Activity

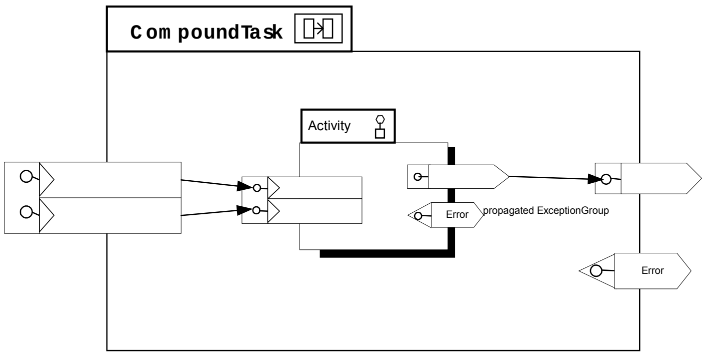


Figure 3-54 An unhandled ExceptionGroup that will be propagated if it is enabled at runtime

Fully Scoped name

ECA::BusinessProcess::ExceptionGroup

Owned by

CompoundTask

Properties

Associated elements

Constraints

3.16.1.12 ProcessRole

Semantics

ProcessRole defines a placeholder for concrete ProcessComponents that perform an Activity or that are used in the performing of an Activity. It defines a placeholder for behavior in a context. ProcessRole is a subtype of ComponentUsage with some qualifying attributes. The owner of a ProcessRole is a CompoundTask and the behavior of the ProcessRole becomes part of the behavior of Activities to which it is associated. The uses association of a ProcessRole (inherited from ComponentUsage) defines the type of ProcessComponent that is required to be bound to the placeholder.

Runtime Semantics: When an Activity is enabled, binding of any associated unbound ProcessRole instances ensues based on the values of the selectionRule and creationRule expressions. Note that some ProcessRole instances may have been bound previously due to an association with another Activity that has already been enabled so no further binding is needed.

If both the selectionRule and creationRule expressions are empty, then it is left up to the Activity itself to perform binding, otherwise binding takes place as follows:

Binding of an unbound ProcessRole begins by determining the *candidate instances*. These are the set of ProcessComponent instances with a compatible type and that satisfy the selectionRule. The selectionRule may refer to the values of the input ProcessPortConnectors of any of the ProcessRole's associated Activities. It is incumbent on the modeler to ensure that the selectionRule is well-formed in the face of attributes that may not yet have values.

If there are no candidate instances, and the creationRule expression is non-empty, it will be used to generate a new candidate instance (or instances if the expression returns multiples).

One of the candidate instances will then be bound to the ProcessRole. If there are no candidate instances, the containing Activity instance will have its system ExceptionGroup enabled.

We note that something akin to the OMG Trader service can be used for this binding process. Also, the bound entity may be a proxy for a person such as a worklist in a workflow execution environment.

Inheritance

ECA::CCA::ComponentUsage

ProcessRole

Fully Scoped name

ECA::BusinessProcess::ProcessRole

Owned by

CompoundTask

Properties

selectionRule

An expression describing the set of entities that may be bound to this ProcessRole.

creationRule

An expression describing how to create a new entity that may be bound to this ProcessRole.

Associated elements

ProcessComponent: The uses association, inherited from ComponentUsage, indicates a type of ProcessComponent (an abstract ProcessComponent). A concrete instance of this type must be bound to the ProcessRole at runtime.

Activity: These may be associated with ProcessRoles by one or more of the following: performedBy and/or usesArtifact and/or responsibleFor.

Constraints

[1] The ProcessComponent at the opposite end of the uses association must be abstract.

3.16.1.13 Performer

Semantics

A Performer ProcessRole is specifically for identifying an Entity that can perform the Activity to which it is associated.

Inheritance

```
ECA::CCA::ComponentUsage
        ProcessRole
                Performer
```

Fully Scoped name

ECA::BusinessProcess::Performer

Owned by

CompoundTask

Properties

Associated elements

Activity: These may be associated with Performers by a performedBy association.

Constraints

[1] A Performer may only be associated with Activities using the performedBy association.

3.16.1.14 Artifact

Semantics

A Performer ProcessRole is specifically for identifying an Entity that is needed by an Activity as a resource.

Inheritance

```
ECA::CCA::ComponentUsage
        ProcessRole
                Artifact
```

Fully Scoped name

ECA::BusinessProcess::Artifact

Owned by

CompoundTask

Properties

Associated elements

Activity: These may be associated with Artifact by a usesArtifact association.

Constraints

[1] An Artifact may only be associated with Activities using the usesArtifact association.

3.16.1.15 ResponsibleParty

Semantics

A ResponsibleParty ProcessRole is specifically for identifying an Entity that has responsibility for the Activity to which it is associated.

Inheritance

ECA::CCA::ComponentUsage
    ProcessRole
        ResponsibleParty

Fully Scoped name

ECA::BusinessProcess::ResponsibleParty

Owned by

CompoundTask

Associated elements

Activity: These may be associated with ResponsibleParties by a responsibleFor association.

Constraints

[1] A ResponsibleParty may only be associated with Activities using the responsibleFor association.

3.17 Notation for Activity and ProcessRole

As shown in Figure 3-56, an Activity is represented similarly to a ProcessComponent. If the Activity uses a CompoundTask that is not primitive (i.e., the Composition is non-empty and the isPrimitive attribute is false), then the ProcessComponent rectangle has a drop-shadow as shown in Figure 3-57.

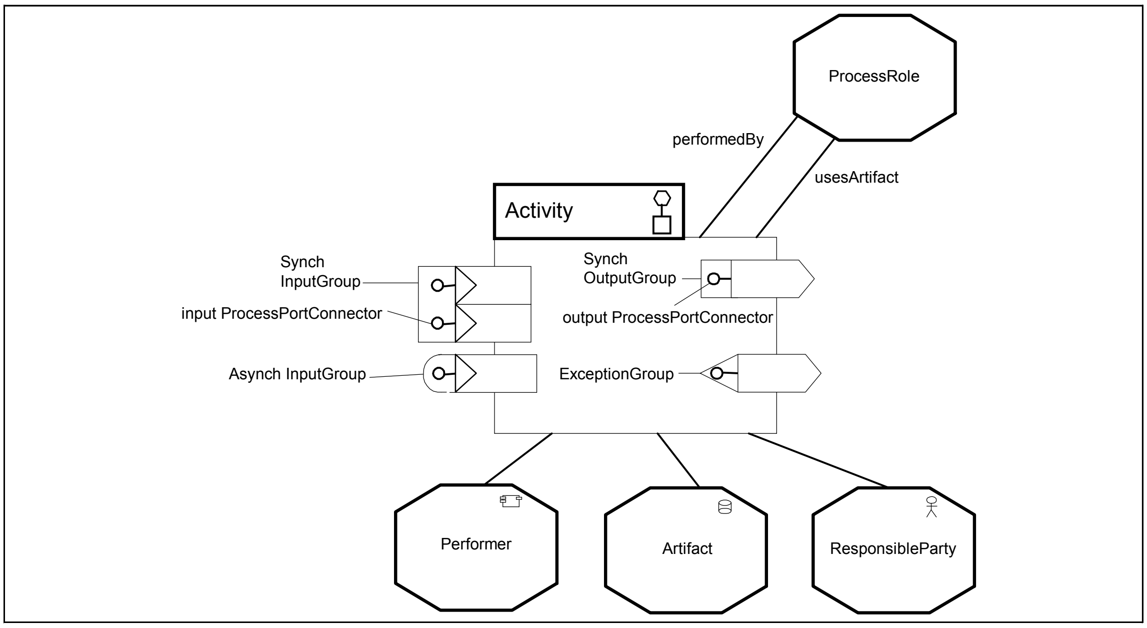


Figure 3-56 Activity with synchronous and asynchronous InputGroups, an OutputGroup and an ExceptionGroup

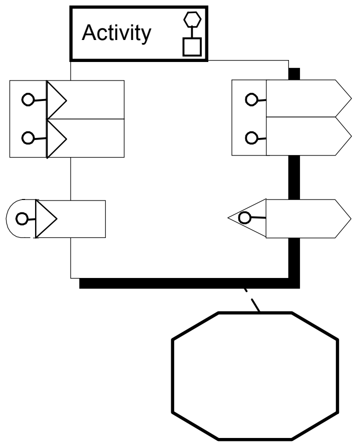


Figure 3-57 Activity that is involves creation of a Composition of nested Activities, etc.

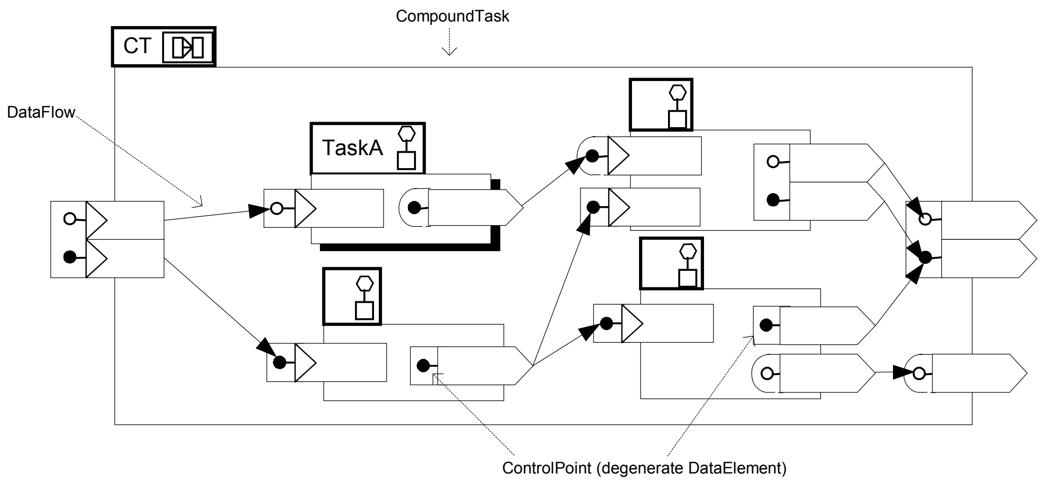


Figure 3-58 A CompoundTask showing its composed Activities

The lollipops represent ProcessFlowPorts and the boxes surrounding them represent ProcessMultiPorts. InputGroups appear on the left-hand side of the Activity and OutputGroups appear on the right-hand side. Rectangular tabs are used to indicate synchronous ProcessMultiPorts, rounded tabs are used to indicate asynchronous ProcessMultiPorts. Triangular or bevel-edged tabs are used to indicate ExceptionGroups, which are a kind of OutputGroup, and hence always appear on the right.

ProcessRoles are drawn as octagons and are associated with Activities by either the performedBy association for Performer roles, the usesArtifact association for Artifact roles, or the responsibleFor association for ResponsibleParty roles. These associations are drawn as a solid line annotated with the association name. See Section 3.16.1.12, “ProcessRole,” on page 3-147 for more detail on the definition and usage of ProcessRoles. It should be noted that a single ProcessRole may be an artifact role in one association and a performer role in another association at the same time. Additionally, an Activity that has a uses association to a CompoundTask with composed Activities, DataFlows and ProcessRoles, may not have a performedBy association to a ProcessRole.

3.18 Process Model Patterns

The rest of this section describes various patterns of common usage and associated special notation that may be useful when using the ECA Process Model. We first describe the pattern in terms of its normal notation, possibly with parameterized parts, and in some cases then provide alternative shorthand notations.

We begin with some simple patterns then move on to more complex patterns involving looping. In general, arbitrary loops in a business process specification can be quite subtle in their behavior, especially in conjunction with concurrent threads. It is for this reason that we restrict an Activity with synchronous DataGroups to executing once only. The looping patterns presented here avoid these problems since they are always defined in terms of an underlying recursive invocation structure.

It should be noted that the UML template notation, and the Patterns Framework introduced in Chapter 4, are not sufficient to express the complexity required by these patterns, since they usually consist of a CompoundTask parameterized by an Activity that will have some unknown number of ProcessMultiPorts and ProcessFlowPorts. When instantiating such a template with respect to a particular Activity, the CompoundTask needs to have corresponding ProcessMultiPorts and ProcessFlowPorts connected by Flows to the equivalent ports on the Activity argument to the template.

3.18.1 Timeout

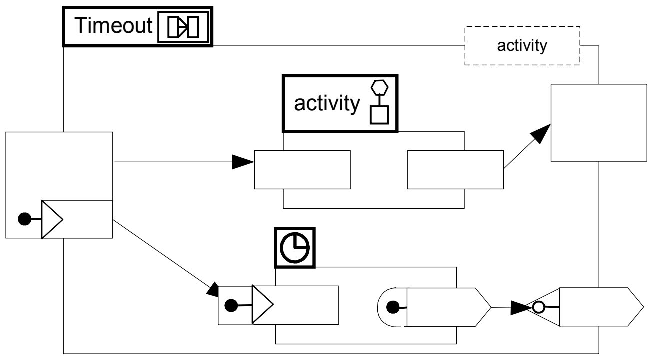


Figure 3-59 Timeout Pattern

Often we will want to have an Activity timeout after some period. The pattern shown in Figure 3-59 illustrates how we might do this. The Activity and timer are started at the same time. If the timer finishes and sends a message on its asynchronous OutputGroup before the Activity finishes, then the ExceptionGroup will be enabled and the CompoundTask will terminate, thus terminating all contained Activities. On the other hand, if the Activity finishes first, the CompoundTask will terminate without waiting for the timer (since it has no synchronous OutputGroups).

A shorthand notation for this pattern is given in Figure 3-60. This notation may also include a duration parameter, or absolute time parameter, which would be provided as input to the underlying timer activity.

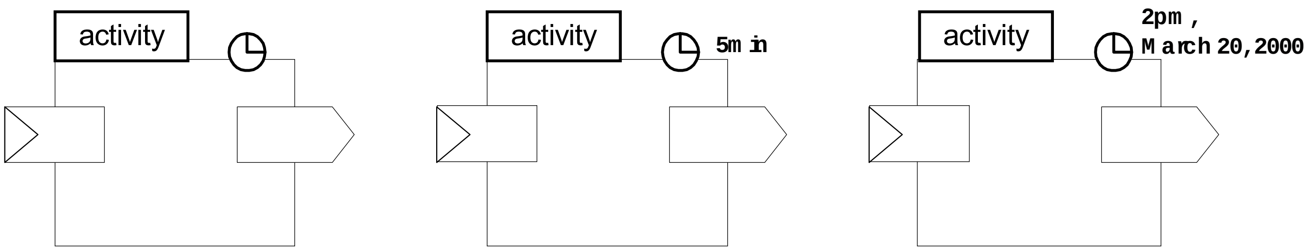


Figure 3-60 Timer pattern notation

Note we do not mandate any particular implementation for the timer task, we merely posit its existence. It would be up to the modeler to have an appropriate performedBy association, or for particular mappings to provide a suitable implementation.

3.18.2 Terminate

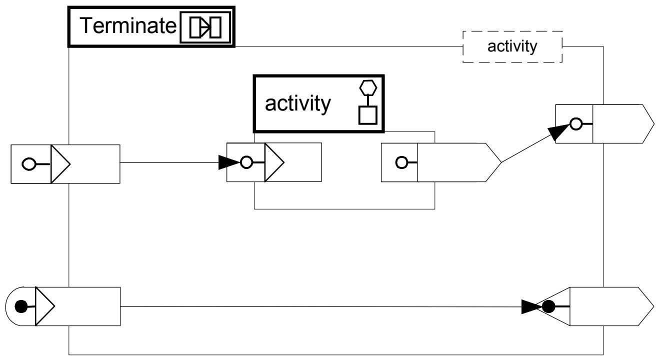


Figure 3-61 Templated activity supporting a terminate message

We may wish to be able to terminate an Activity before it has completed of its own accord. The pattern shown in Figure 3-61 illustrates how an Activity can be wrapped to support an additional asynchronous InputGroup that, on reception of a message, will result in the activity being terminated and an exception being thrown.

That is, if a message is sent to the asynchronous InputGroup of the CompoundTask, then it will immediately flow to the CompoundTask's ExceptionGroup causing the CompoundTask to terminate, thus terminating the contained Activity.

There is no suggested shorthand notation for this pattern. However, tools may wish to support the implicit inclusion of an appropriately labeled asynchronous InputGroup and corresponding ExceptionGroup on any arbitrary Activity.

3.18.3 Activity Preconditions and Activity Postconditions

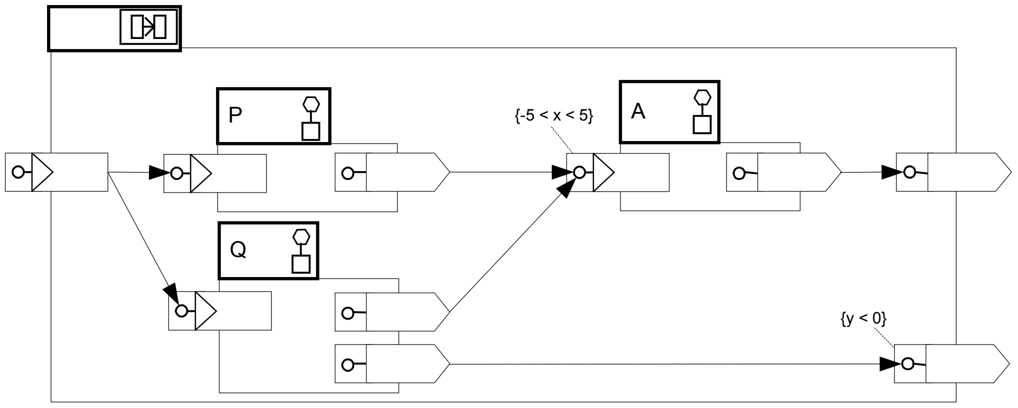


Figure 3-62 Preconditions on an InputGroup and an OutputGroup

Sometimes it may be desirable to add a precondition to the InputGroup of an Activity, or the OutputGroup of a CompoundTask, to further constrain the enabling of the InputGroup/ OutputGroup. For example, there may be multiple DataFlows to an input, but we wish to ignore any values that fall outside a given range. Figure 3-62 illustrates how one might attach such a guard constraint where x and y are attributes of the DataGroup (or perhaps even attributes of their contained DataElements).

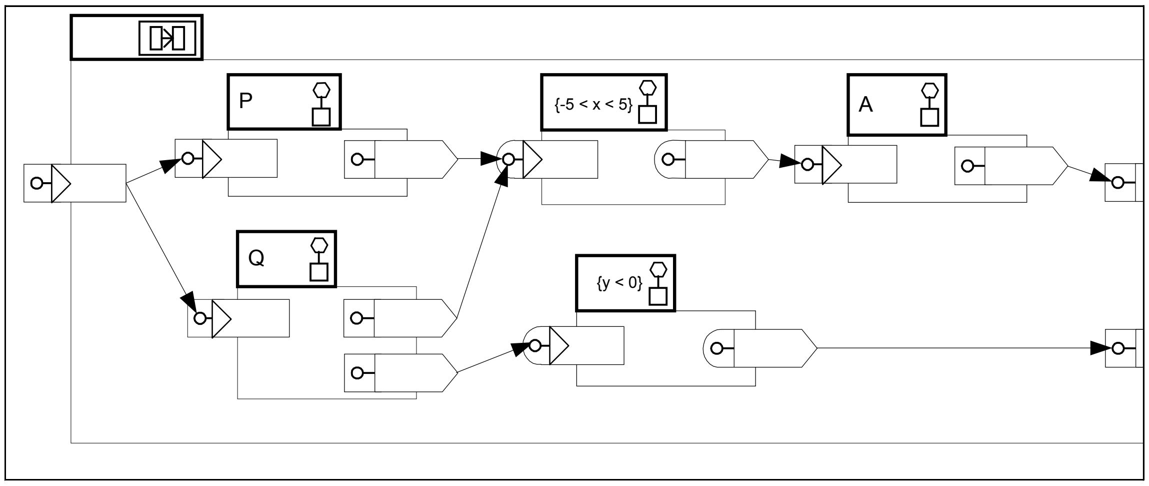


Figure 3-63 An equivalent model to that of Figure 3-62 using condition tasks

Figure 3-63 shows an equivalent CompoundTask to that of Figure 3-62 but using explicit filter Activities.

If a filter Activity does not produce enough outputs to satisfy the multiplicity requirements of the Activity it is guarding, then the Activity will not start. As can be seen from Figure 3-63, if neither filter is satisfied, then Activity ‘A’ will never run, so the CompoundTask instance will satisfy its completion criteria (quiescence) without either OutputGroup being satisfied, which causes its system ExceptionGroup to be enabled.

In a similar way, we may also attach a post-condition to an Activity's OutputGroup to ensure that the result of the Activity satisfies some condition. This is shown in Figure 3-64.

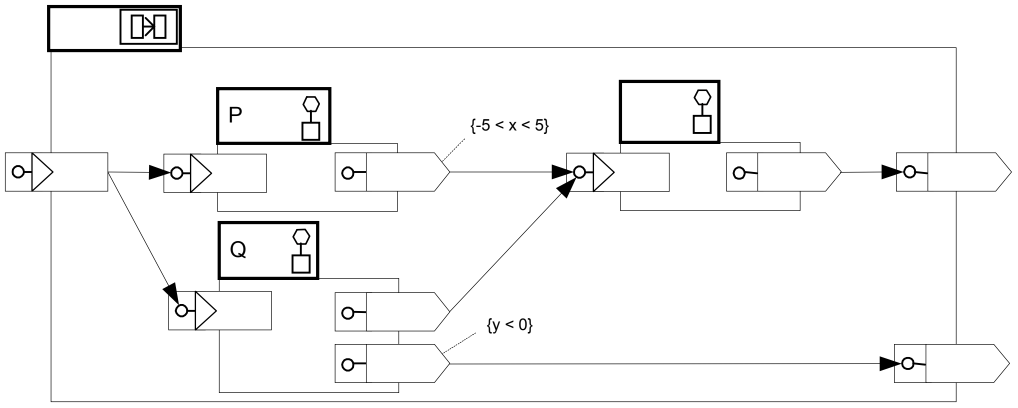


Figure 3-64 Post-conditions on OutputGroups of Activities

Figure 3-65 shows an equivalent CompoundTask to that of Figure 3-64 but using explicit filter Activities that have a 'success' OutputGroup and a 'fail' ExceptionGroup. Thus, if the postcondition does not hold, the CompoundTask's system ExceptionGroup will be enabled.

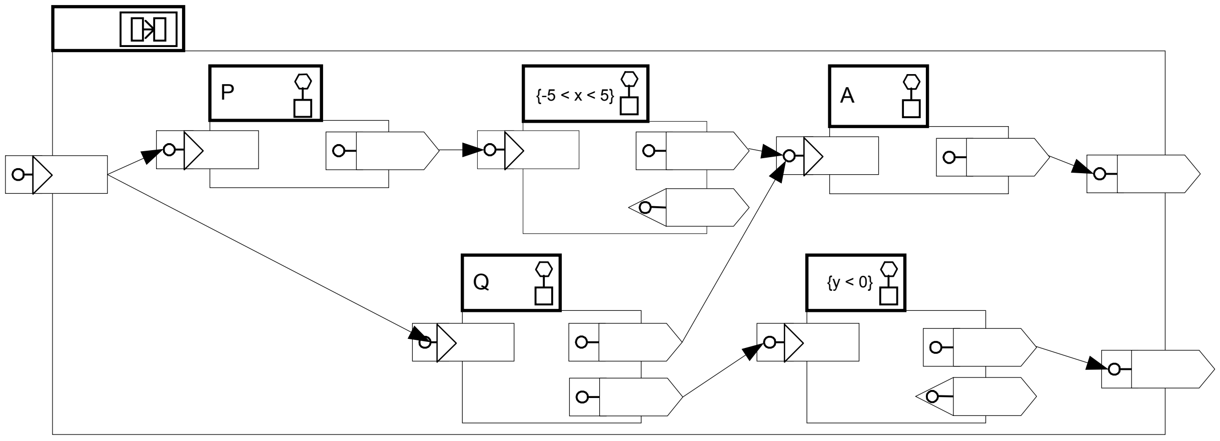


Figure 3-65 An equivalent model to that of Figure 3-64, using condition tasks

3.18.4 Simple Loop

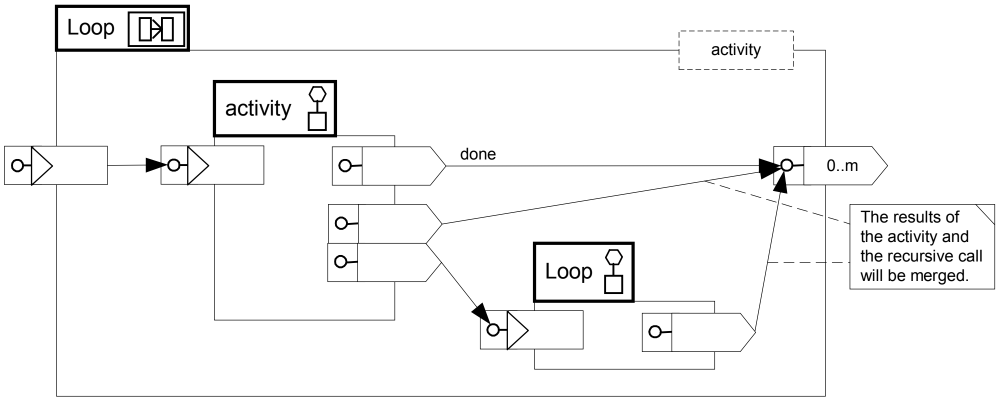

Loop
activity
activity
done
0..m
The results of the activity and the recursive call will be merged.
Loop

Figure 3-66 Simple Loop Pattern

The pattern shown in Figure 3-66 shows how we might repeatedly invoke an Activity until a particular OutputGroup is enabled. If the cardinality of the Output in the loop CompoundTask is 0..*, then all the results of the Activity will be collected. If it is 0..m for some finite m, then some subset of those results will be collected.

In this case, we assume that the exit condition and the loop action are combined into a single Activity, possibly via a CompoundTask. Normally this will not be the case, however, and the more general patterns described in Section 3.18.5, "While and Repeat-Until Loops," on page 3-158 through Section 3.18.7, "Multi-Task," on page 3-160 will be used.

A special-case shorthand notation for such a loop is shown in Figure 3-67. The looping flow indicates that simple recursion is taking place. Any OutputGroup containing a ProcessPortConnector that is the source of a looping flow may only be the source of flows to a single InputGroup.

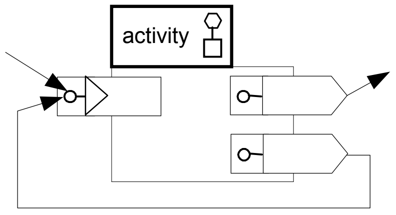


Figure 3-67 Simple Loop Notation

3.18.5 While and Repeat-Until Loops

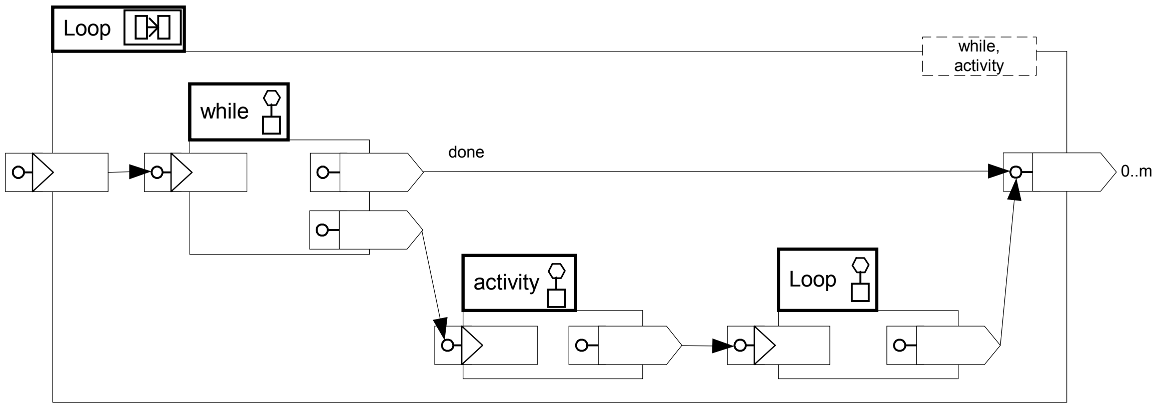


Figure 3-68 While Loop Pattern

In Figure 3-68 we see a more general ‘while’ loop pattern with separate exit test and loop body, and Figure 3-69 shows a slightly different pattern that results in a ‘repeat-until’ loop. The ‘while’ and ‘until’ Activities represent some kind of boolean expression evaluation engine.

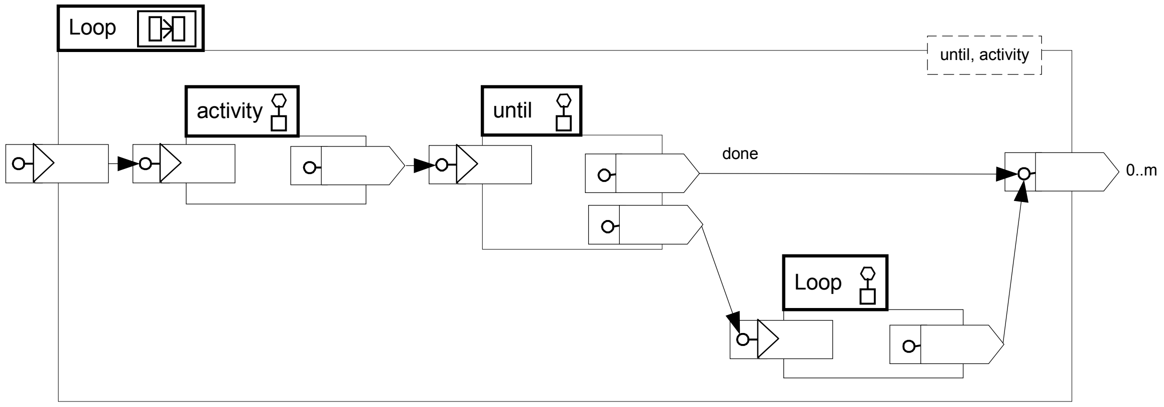


Figure 3-69 Repeat/Until Loop Pattern

As for the Simple Loop, these loops could be drawn as shown in Figure 3-70 and Figure 3-71 respectively.

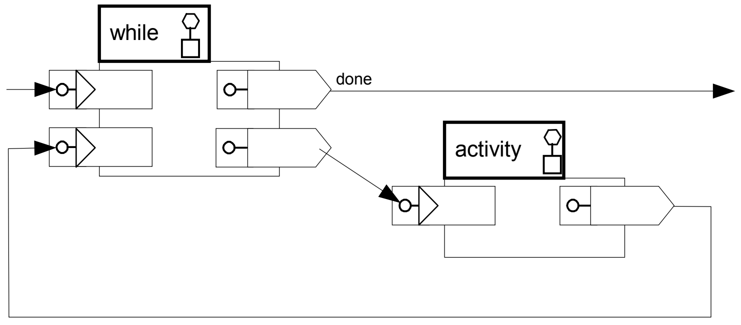


Figure 3-70 While Loop Notation

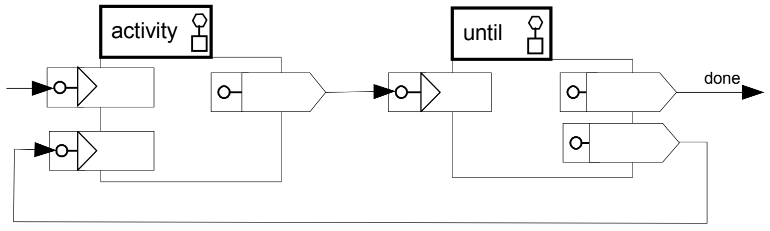


Figure 3-71 Repeat-Until Notation

3.18.6 For Loop

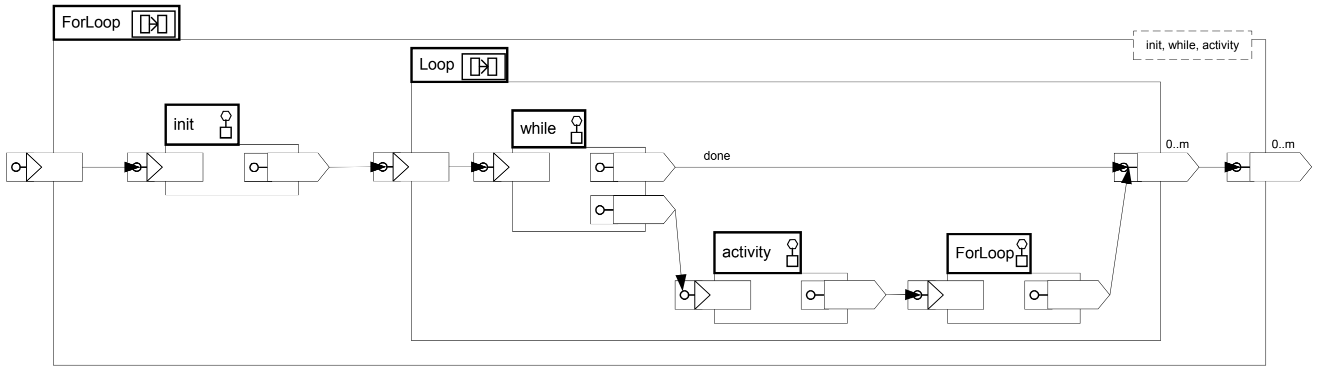


Figure 3-72 For Loop Pattern

The pattern in Figure 3-72 shows how to do a for-loop with a generalized initialization step, loop test, and loop body as popularized by the C, C++, and Java languages. Note that the inner loop is the while-loop pattern and hence the special-case notation for while-loops can be used.

3.18.7 Multi-Task

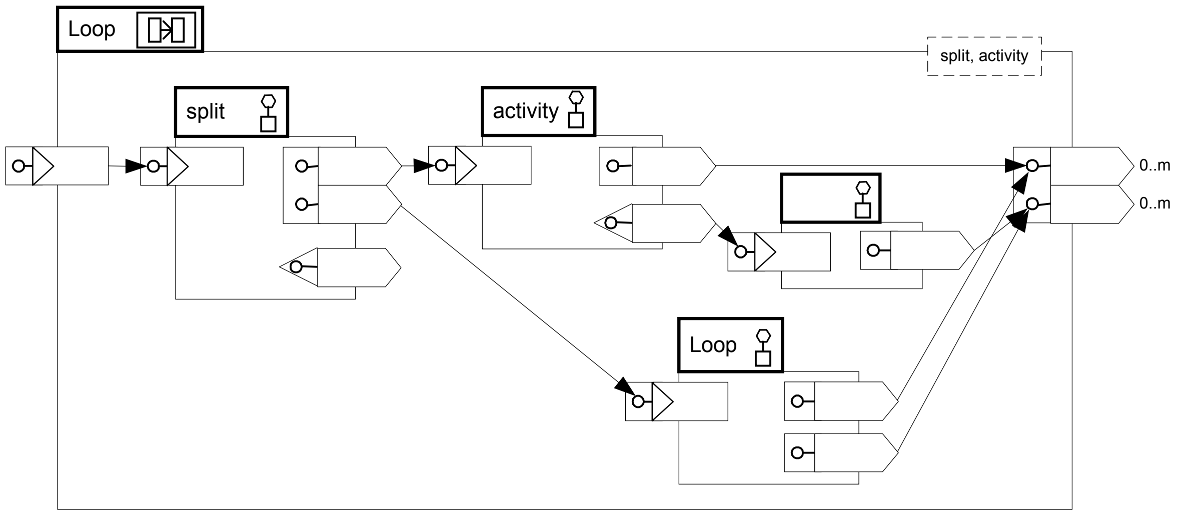


Figure 3-73 Pattern for a multi-task

The pattern in Figure 3-73 shows how to process a collection of items in parallel and collect the results. The split activity takes a collection of items and splits them into a head and a tail. The head is passed to the activity for processing, while a concurrent recursive invocation of the loop is initiated to process the tail. If, however, the collection is empty, then the split's other OutputGroup is enabled and the loop CompoundTask finishes. No explicit flow from this OutputGroup to the CompoundTask's OutputGroup is required since all its Outputs will be satisfied with a zero cardinality.

Intuitively, what happens when this pattern executes is as follows. When a collection of items is passed in to the multi-task pattern, a set of concurrent loops and activities is spawned, one pair for each item in the collection. The activity processes an item, and the concurrent loop recursively handles the other n-1 items.

Note that if an Activity processing an item throws an exception, it is caught and passed to a second Output in the OutputGroup. This means that a single failed Activity doesn't cause all the other Activities to be terminated and the completed activities to throw away their results. This is especially useful in the case where we might wish to apply the timer pattern to the Activity.

No shorthand notation for multitask is suggested.

3.19 Full Model

The diagram below represents the full metamodel for the Business Process model.

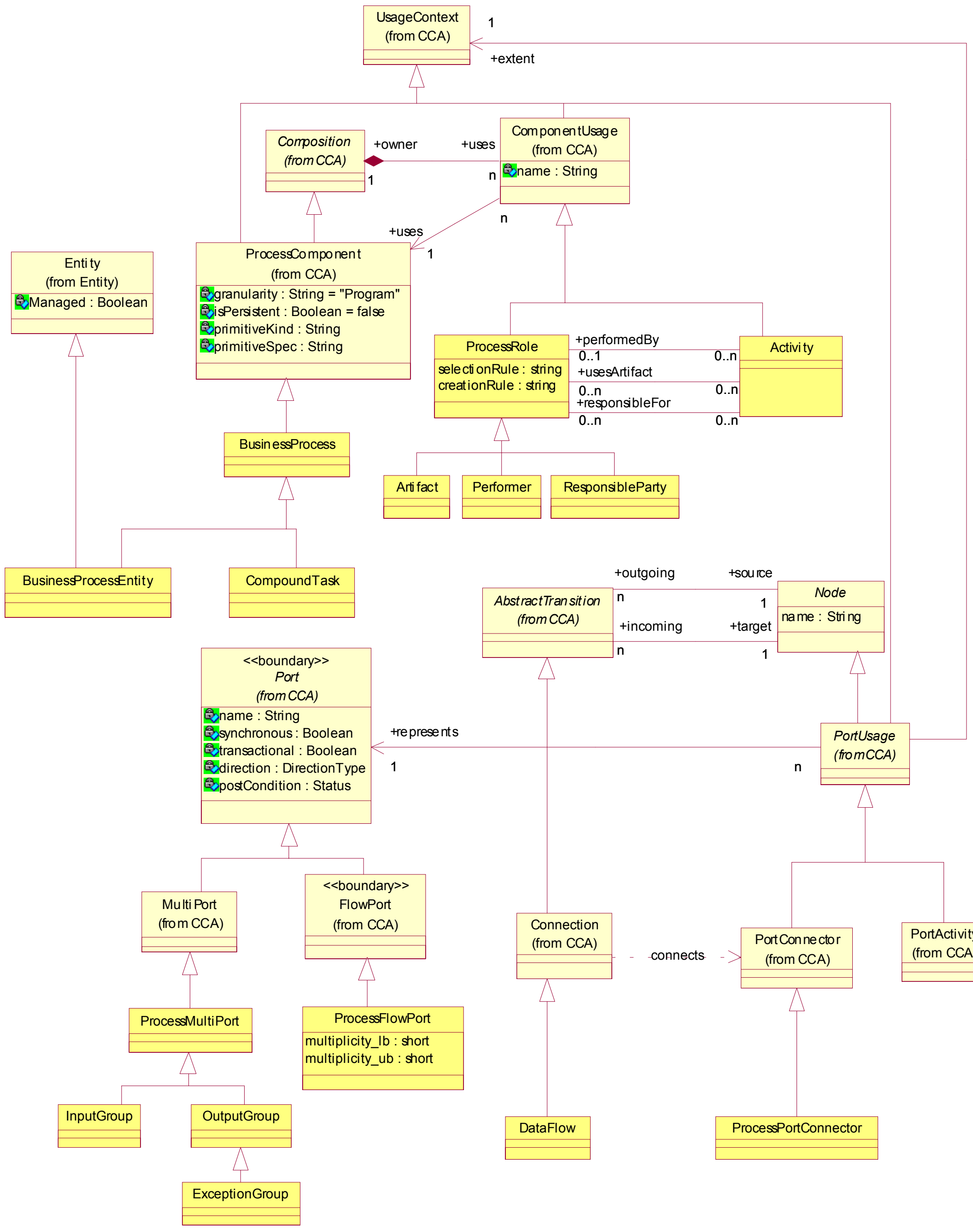


Figure 3-74 Combined MOF model of Process

References A

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] OMG, Model Driven Architecture – under development

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

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

Glossary

The Glossary defines the specialist terms used in this specification.

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

Index

Index