

---

# UML Profile for EDOC

---

## **Submitted by:**

**Cooperative Research Centre for Enterprise Distributed Systems Technology  
(DSTC)**

**International Business Machines (IBM)**

## **Supported by:**

**University of Newcastle upon Tyne (UK)**

---

Copyright 2001, DSTC Pty Ltd (Cooperative Research Centre for Enterprise Distributed Systems Technology), International Business Machines.

DSTC Pty Ltd and IBM hereby grant to the Object Management Group, Inc. a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version.

Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conforming any computer software to the specification.

## NOTICE

The information contained in this document is subject to change without notice.

The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any companies' products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND DSTC MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The aforementioned copyright holders shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means—graphic, electronic or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

OMG and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB CORBA, CORBAfacilities, CORBAservices, and UML are trademarks of the Object Management Group.

## ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by sending email to [issues@omg.org](mailto:issues@omg.org). Please reference precise page and section numbers, and state the specification name, version number, and revision date as they appear on the front page, along with a brief description of the problem. You will not receive any reply, but your report will be referred to the OMG Revision Task Force responsible for the maintenance of the specification. If you wish to be consulted or informed during the resolution of the submitted issue, indicate this in your email. Please note that issues appear eventually in the issues database, which is publicly accessible.

---

---

---

<b>1 Overview.....</b>	<b>9</b>
1.1 Introduction.....	9
1.2 Submission Contact Points .....	9
1.3 Structure of This Submission.....	10
1.4 RFP Requirements .....	10
1.5 Design Philosophy .....	11
1.5.1 Using UML 1.3 for EDOC Modeling.....	11
1.5.2 Business Perspective to Process Modeling for Enterprise.....	12
1.5.3 Business Perspective to Roles and Events in the Enterprise.....	13
1.5.4 Trade-off between Complexity of Models and Notational Convenience .....	13
1.6 Technical Overview .....	14
1.6.1 Business Process Model.....	14
1.6.2 Business Role Model .....	16
1.6.3 Business Event Model.....	17
1.7 The Relationships between Models .....	18
1.7.1 Entity Modeling .....	18
1.7.2 Mapping Process aspects to Events .....	19
1.7.3 Mapping to the EAI Profile.....	19
1.7.4 Mapping to CORBA and Services.....	19
1.8 Acknowledgments.....	19
<b>2 Business Process Model.....</b>	<b>21</b>
2.1 Overview.....	21
2.2 Task (abstract).....	23
2.3 DataGroup (abstract).....	25
2.4 DataElement (abstract).....	26
2.5 InputGroup.....	27
2.6 Input .....	28
2.7 OutputGroup .....	30
2.8 Output .....	31
2.9 ExceptionGroup.....	32
2.10 Activity .....	35
2.11 CompoundTask .....	36
2.12 BusinessProcess .....	38
2.13 DataFlow.....	38
2.14 DataMap.....	40
2.15 BPRole .....	41
2.16 Runtime Semantics Summary.....	43
<b>3 Business Event Model.....</b>	<b>45</b>
3.1 Introduction & Overview.....	45

# Table of Contents

---

3.2	EventTransceiver (abstract) .....	46
3.3	EventSource .....	48
3.4	EventSink .....	49
3.5	Actions at Sources .....	50
3.5.1	Model Element .....	50
3.5.2	Activity .....	51
3.5.3	DataGroup .....	51
3.5.4	DataElement .....	51
3.5.5	BPRole .....	52
<b>4 Notation and Patterns .....</b>		<b>53</b>
4.1	Introduction .....	53
4.2	Activity and BPRole .....	53
4.3	CompoundTask .....	55
4.4	DataMap .....	56
4.5	Events .....	57
4.6	Notation for Event Sources and EventSinks .....	58
4.7	Patterns Overview .....	59
4.8	Timeout .....	59
4.9	Terminate .....	60
4.10	Guards .....	61
4.11	Simple Loop .....	62
4.12	While and Repeat-Until Loops .....	63
4.13	For Loop .....	64
4.14	MultiTask .....	64
<b>Appendix A Example EDOC Specification .....</b>		<b>67</b>
A.1	An Example Specification of an Enterprise System .....	67
A.2	The Procurement System - An Informal Description .....	67
A.3	The Business Process Model .....	68
A.4	Detailed Task Description .....	69
A.4.1	Sourcing and Sourcing Freight-Dependent Request Processes .....	69
A.4.2	Evaluation .....	69
A.4.3	Award .....	71
A.4.4	Maintain .....	71
A.4.5	Release .....	71
A.4.6	Monitor .....	71
A.4.7	Process Order .....	72
A.4.8	Receipt and Approve .....	72

---

<b>Appendix B Model to Model Mappings.....</b>	<b>73</b>
B.1 EAI Mapping for the Business Process Model .....	73
B.1.1 BPRole .....	74
B.1.2 Activity .....	76
B.1.3 DataElement.....	76
B.1.4 Input .....	76
B.1.5 Output .....	76
B.1.6 Control Point.....	77
B.1.7 DataGroup.....	77
B.1.8 ExceptionGroup .....	80
B.1.9 DataFlow .....	80
B.1.10 CompoundTask .....	82
B.2 Process Model to Entity Model Mapping .....	84
B.2.1 Generic Mapping of DataGroup to Operation .....	84
B.2.2 Mapping Synchronous InputGroups .....	85
B.2.3 Mapping Synchronous OutputGroups .....	87
B.2.4 Mapping Asynchronous OutputGroups .....	88
B.2.5 Mapping Asynchronous InputGroups.....	88
B.2.6 Mapping a Business Process as an Entity .....	89
B.2.7 Generation of an Activity from Features of an Entity .....	90
B.2.8 Generation of Skeleton Collaboration from an Activity.....	91
B.3 Mapping Asynchronous DataGroups to Event Transceivers.....	93
B.3.1 Asynch InputGroup Flows to Asynch OutputGroup .....	95
B.3.2 Asynch InputGroup Flows to Asynch InputGroup.....	95
B.3.3 Asynch Input Flows to Synch Output.....	96
B.3.4 Synch Input Flows to Asynch Out Port .....	97
<b>Appendix C Mapping to CORBA and Services.....</b>	<b>99</b>
C.1 Introduction.....	99
C.1.1 Alternative Mappings.....	99
C.1.2 Structure.....	100
C.2 Common Base Types for the Business Process Model.....	100
C.2.1 Task (abstract).....	100
C.2.2 BusinessProcess .....	100
C.2.3 CompoundTask.....	100
C.2.4 Activity .....	101
C.2.5 DataMap.....	102
C.2.6 BPRole .....	102
C.3 Notification-based Mapping for the Business Process Model.....	103
C.3.1 Task (abstract).....	104
C.3.2 DataElement (abstract).....	104
C.3.3 CompoundTask.....	105

# Table of Contents

---

C.3.4	ExceptionGroup .....	105
C.4	Interface-based Mapping for the Business Process Model .....	105
C.4.1	Task (abstract).....	106
C.4.2	DataGroup (abstract).....	107
C.4.3	DataElement (abstract).....	108
C.4.4	CompoundTask.....	108
C.4.5	ExceptionGroup .....	108
C.4.6	BusinessProcess .....	109
C.5	The Business Event Model .....	109
C.5.1	Event Type .....	109
C.5.2	EventTransceiver (abstract) .....	109
C.5.3	EventSource .....	110
C.5.4	EventSink.....	112
C.5.5	Mapping Event Aspects of Specific Elements.....	113
<b>Appendix D Activity Diagram Notation .....</b>		<b>115</b>
D.1	EDOC ModelModeling concepts and Activity Diagram notation.....	115
<b>Appendix E MOF Models .....</b>		<b>121</b>
E.1	Other Relevant Documents .....	121

## 1.1 Introduction

The Cooperative Research Centre for Enterprise Distributed Systems Technology (DSTC) and International Business Machines (IBM) are pleased to submit our response to the Enterprise Distributed Object Computing (EDOC) RFP. DSTC is actively pursuing a research and development programme in the application of distributed systems technology for enterprise systems and is delighted to contribute this expertise to the adoption of an EDOC specification for the OMG. IBM has contributed expertise from its wealth of experience in Enterprise Middleware, and the recent development of models for systems with tools that can generate implementation skeletons in multiple platforms.

This EDOC specification provides the following benefits:

- **a small but powerful set of EDOC modeling concepts**
- **an expressive graphical notation**
- **a set of powerful patterns that enhance the standard model concepts**
- **a set of model to model mappings including one to EAI-based enterprise systems**
- **a mapping basis for automatic generation of CORBA-based enterprise systems**

## 1.2 Submission Contact Points

Feedback on this submission is most welcome, and should be directed to:

DSTC EDOC team, [edoc-rfp1@dstc.edu.au](mailto:edoc-rfp1@dstc.edu.au)

Dr Michael Lawley, [lawley@dstc.edu.au](mailto:lawley@dstc.edu.au)

Mr Keith Duddy, [dud@dstc.edu.au](mailto:dud@dstc.edu.au)

Dr Zoran Milosevic, [zoran@dstc.edu.au](mailto:zoran@dstc.edu.au)

CRC for Enterprise Distributed Systems Technology (DSTC)  
Level 7, GP South  
The University of Queensland  
Brisbane, QLD 4072  
Australia

Phone: +61 7 3365 4310  
Fax: +61 7 3365 4311  
WWW: [www.dstc.edu.au](http://www.dstc.edu.au)

Mr Marc-Thomas Schmidt, [mts@uk.ibm.com](mailto:mts@uk.ibm.com)

International Business Machines (IBM)  
Hursley House  
Hursley Park  
Winchester  
Hants SO21 2JN  
UK

Phone: 01962 815308  
Fax: 01962 842237  
WWW:[www.ibm.com](http://www.ibm.com)

## 1.3 Structure of This Submission

The content of this submission is structured as follows:

- an overview of the submission (Section 1)
- business process and role model (Section 2)
- business event model (Section 3)
- notation for the above models and patterns of use for process models (Section 4)
- an example using our models and notation (Appendix A)
- mappings from EDOC models to other models (including the draft UML Profile for EAI) (Appendix B)
- mappings from EDOC models to CORBA technology (Appendix C)
- using Activity Diagrams as an alternative process model notation (Appendix D)

## 1.4 RFP Requirements

As the structure above indicates, this submission focuses on providing models with detailed semantics for:

- business processes,
- business entities (as roles denoting collections of entities) and,

- business events,

together with a non-normative mapping of these models to CORBA and EAI technology. Note that business rules are implicitly incorporated in the other three models.

Collectively, this material addresses the following RFP requirements:

- Modeling of Business Process, Entity, Rule, and Event Objects
- Proof of Concept of Mappability

This submission is aligned with the Meta-Object Facility, as the models and their well-formedness rules can be represented using the MOF.

## 1.5 Design Philosophy

### 1.5.1 Using UML 1.3 for EDOC Modeling

This submission provides models for Enterprise Distributed Object Computing (EDOC). These models are based on existing UML modeling concepts and define a set of new modeling elements and relationships between them.

In deriving these models, our starting point was an analysis of how the EDOC requirements stated in this RFP can be met using existing UML modeling concepts.

We have found that:

- there is a need for further extensions and refinement of the existing concepts to better meet *enterprise* requirements such as:
  - better support for capturing the coordination and compositional semantics of business processes and their constituent parts, including a possibility for reusing off-the-shelf business process components.
  - explicit support for business events and exceptional situations in a business process.
  - support for declarative specification of abstract behaviour for those business entities that can execute (or are used by) business process so that the corresponding business process elements can be dynamically bound to such entities.
- in addition to the expressiveness and power of EDOC modeling concepts needed for enterprise systems specification, they also need to be formal enough to allow model transformation from the enterprise model into various implementation choices.

The above points mean that it is non-trivial for an enterprise modeler to effectively and efficiently use UML 1.3. for modeling systems to support Enterprise Distributed Object Computing. We believe that the extension of the UML concepts as per above will further augment the richness of the UML in terms of its computational expressiveness.

To address these points and provide a set of self-contained concepts suitable for practical enterprise modeling we have utilised the ideas from various areas relevant to Enterprise Distributed Object Computing. This includes areas such as workflow systems, requirements engineering, and the ODP Enterprise Language Standard.

## 1.5.2 Business Perspective to Process Modeling for Enterprise

Our approach to modeling *business processes* is based on our understanding of what are the critical *business* issues to be addressed when describing and implementing processes in the enterprise.

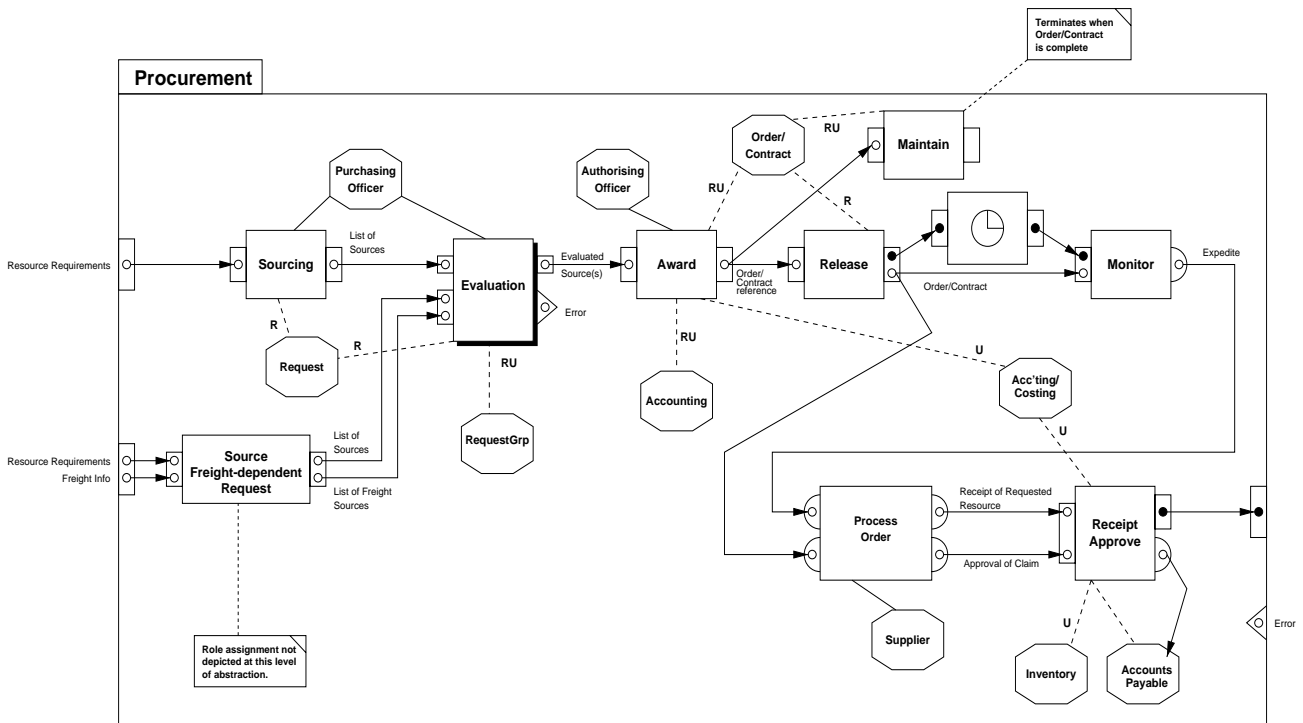


Figure 1-1 An Example Business Process.

As illustrated in Figure 1-1, a business process is represented as a dependency graph of business tasks composed in a specific way to achieve some particular objective. There are many possible ways how these tasks can be composed in enterprise. Examples are the use of synchronous flows between tasks to specify causality between tasks (i.e. data flows for passing data between tasks or control flows when there are no data), concurrency and parallelism of various 'threads' in a business process and the use of asynchronous business events that can be integrated with the synchronous execution of tasks in a business process. In addition, there is a need to specify points of interactions between business process and the rest of enterprise and we do this by specifying abstract behaviour for the entities that execute certain parts of a business process or are

used by them. Our model provides a rich semantics for expressions of these business issues and also supports the composition of business tasks in a way that is suitable for implementation as off-the-shelf components.

We believe that this RFP calls for an explicit representation of the business semantics of business processes and this was our starting premise. Although we recognise that there is certain overlap with many workflow products, we view them as an IT solution to automating and managing business processes, mostly focussing on the execution semantics. Instead, we have attempted to come up with a minimal business process model that encompasses a number of workflow execution semantics. In addition, we consider business processes in the context of other business determinants, such as business roles, business entities and business events resulting in an emphasis on business semantics over computational semantics. However, our non-normative mappings do demonstrate ways of implementing these business process concepts. For example we show how our EDOC modeling concepts can be implemented using CORBA interfaces including the OMG's Workflow Management Facility specification and also using a higher level implementation choice such as UML profile for EAI.

### *1.5.3 Business Perspective to Roles and Events in the Enterprise*

We believe that business process modeling is only one (though frequent) approach to modeling a business. There are other alternative and/or complementary ways of modeling such as *business role* modeling. This submission supports modeling of those roles that are relevant in the context of business process and we refer to these as Business Process roles (or BPRoles for short). These are expressed in terms of an abstract specification of either business entities to perform execution of tasks (referred to informally as performer roles) or entities to be used by tasks (referred to informally as artifact roles).

The separation of the notion of BPRole and business entity provides a powerful mechanism for distinguishing between required behaviour and business entities that can satisfy this behaviour in the context of a business process. One business entity can fill more than one role and one role can be filled by different entities, as long as behaviour of such an entity is compatible with the behaviour of that business role. We note that the core ideas of our business role modeling in terms of an organisational structure are now taken out of this submission as they are more with scope of the ongoing Organisational Structure OMG standardisation.

In both process-based and role-based approaches, it is important to expose *business events* of significance for the enterprise. These events are associated with the modeling elements that can be their sources or sinks and our approach allows for flexible mapping of business event parameters onto the business process elements and also business roles.

### *1.5.4 Trade-off between Complexity of Models and Notational Convenience*

In providing notation for our EDOC modeling concepts we have attempted to use 'user-oriented' presentation elements to closely resemble the semantics of particular modeling element. We provide a separate section with a list of these notation elements.

Based on our experience with using our modeling concepts, we anticipate that a number of frequently-used patterns will occur in business modeling, a number of which are presented, in a non-normative fashion, in Section 4.7 onwards. The notation and semantics of these and other patterns may be the subject of further standardisation.

## 1.6 Technical Overview

This specification introduces two main models: the Business Process model and the Event model. Note that the role-based model is defined in the same package as Process model. This role model bridges Process model and the Entity models (those that designers produce using Class Diagrams).

### 1.6.1 Business Process Model

#### Overview

The root concept, a Business Process is defined by a CompoundTask. A CompoundTask comprises a set of Activities, how they communicate, and their synchronization constraints. Some of these Activities may be further decomposed, as described by other associated CompoundTasks. Those not further decomposed, the *leaf* Activities, are performed by entities selected via associated BPRoles.

In particular, each CompoundTask contains:

- a number *InputGroups* and *OutputGroups* representing alternative initializations and terminations of the task
- *Activities*, which are units of work, also with *InputGroups* and *OutputGroups* representing alternative initializations and terminations of these units of work
- *Flows*
  - between the *Inputs* (correlated by *InputGroups*) of the CompoundTask and the *Inputs* of the *Activities*
  - between the *Outputs* (correlated by *OutputGroups*) of *Activities* and the *Inputs* of other *Activities*
  - between the *Outputs* of *Activities* and the *Outputs* of the CompoundTask
- Business Process Roles (*BPRoles*) which indicate the type and manner of selection of objects that may perform, or act as assisting artifacts to, one or more *Activities*

Figure 1-2 illustrates these modeling concepts diagrammatically.

The leaves of the tree are those *Activities* that are enacted by objects represented by *BPRoles*, linked to the *Activities* by *performedBy* associations. *BPRoles* linked by *uses* associations to *Activities* represent artifact objects to assist the performer objects.

The intermediate nodes of the Business Process tree are those *Activities* that are enacted by other CompoundTask definitions.

The InputGroups of CompoundTasks and Activities represent the set of Inputs which require data values in order that the Activity or CompoundTask may execute. They can be seen as correlators for incoming values from several Flows. An InputGroup is analogous to a set of method parameter slots which all require argument values before the method can be meaningfully executed. Likewise, the OutputGroups are sets of Outputs that require values from Flows connected to them in order to complete an execution, like a set of output parameters needed to return from a method call. InputGroups and OutputGroups can be characterized as synchronous (indicating that they require all input/output values to be delivered for task to start/end) or asynchronous (indicating that a complete set of data values in the group may be delivered into or out of an during its execution).

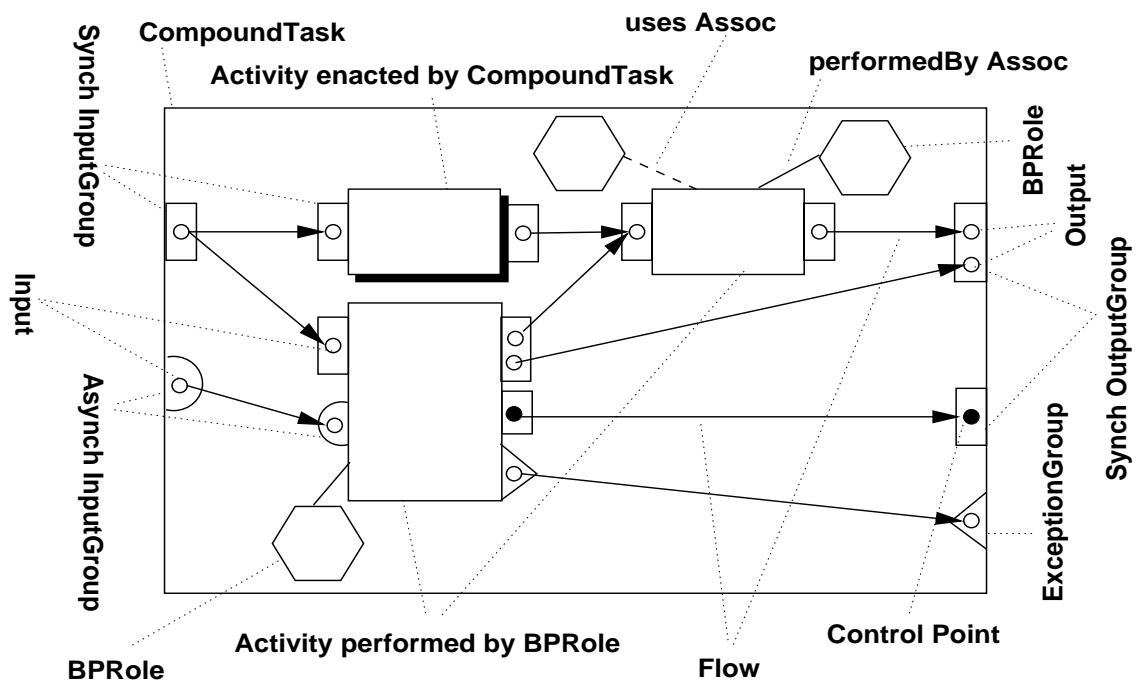


Figure 1-2 An example of the use of EDOC Process modeling concepts

An Activity with at least one synchronous InputGroup may begin execution only once. This is the basis for a clean model for termination of CompoundTasks that contain these Activities. During its execution it may receive sets of values from its asynchronous InputGroups, and send out sets of values from its asynchronous OutputGroups. An Activity with only asynchronous InputGroups starts execution as soon as its containing CompoundTask is enabled, and an Activity with only asynchronous OutputGroups will terminate when all the synchronous Activities in its CompoundTask have finished, and the CompoundTask terminates. A special kind of OutputGroup, the *ExceptionGroup*, allows explicit modeling of error conditions.

Although synchronous Activities may execute only once, repetition of the same Task a fixed number of times may be modeled as several Activities enacting the same CompoundTask definition. In addition, iterations and variable numbers of multiple parallel enactments of the same CompoundTask can be modeled using recursion (an Activity in a CompoundTask is enacted by its parent). This gives a clean extension of the termination semantics for Activities, as each recursive call is to a new Activity which still only executes once, but may re-use the same BPRole to do the work. However, recursion may not be a natural paradigm for many process modelers to use. Therefore, a number of standard patterns for iteration and multiple parallel execution are given which allow modelers to think in terms of looping and thread spawning without introducing any confusion in the termination semantics of Activities which looping tends to do in other process models.

### *Flow-based vs Rule-based Process Definitions*

The process model is oriented toward explicit data and control flow dependencies between Activities. However, some process modelers prefer rule-based specification of Activity enabling. Patterns are also used to present a convenient model for rule-based execution of Activities, such as preconditions and postconditions on Activities.

### *Other Issues*

The Activity enactment semantics within a CompoundTask permit simple grouping of Tasks that have no causal relationships; these are then assumed to execute in parallel. CompoundTasks may also be used simply as the grouping of BPRoles to provide a context within which the Roles' bindings are made, without reference to their explicit invocation through an Activity.

The model permits top-down or bottom-up development of Business Process Definitions. In some cases both approaches are used, and mismatches can occur in the middle. This can be due to an attempt to re-use some existing CompoundTask definition to enact an Activity in another Business Process, or the use of an automatically generated Activity which encapsulates the methods of a Class (see section 1.7). This problem is solved by allowing any Activity that is enacted by a CompoundTask to define *DataMaps* that transform the Inputs of an InputGroup belonging to the Activity into a form usable by the CompoundTask that enacts the Activity. Similarly, the Outputs of the CompoundTask may be transformed by other DataMaps into a form that the Activity can use in its Outputs.

A Business Process Object is an instance of a Class generated automatically from a CompoundTask definition. It represents the object encapsulation of the mechanism by which an instance of this CompoundTask definition will be created and given initialization arguments. (See section 1.7.)

## *1.6.2 Business Role Model*

We use Business Process Roles (BPRoles for short) as a way of describing which objects are capable of performing the Activities in a Business Process, and what the Activities will need to get their work done. BPRoles are associated with Activities by

either a “performedBy” link, or a “uses” link. This distinguishes the BPRoles that denote objects which do the work of the Activity and the those needed by this object for access to information or other arbitrary reasons. Each BPRole identifies a Classifier that is the type of object that will be bound to the BPRole, as well as two expressions that describe how the set of valid objects for the binding may be located, or if this set is empty, how a new object may be created to fill the BPRole.

The Class Model in UML provides a good basis for specification of object types and static relationships between them. However, the associated UML models for interactions between objects (Collaborations and Sequence Diagrams), scope these interactions either at the level of entire classes of objects, or in terms of specific instances. The BPRole concept introduced in this specification allows for a design in which a constrained set of objects of a given class can be specified using some criteria.

The way in which an Activity uses the behavioural features of the object bound to a BPRole is expressed through a Collaboration. This specification maps the InputGroups and OutputGroups of the Activity as methods of classes which are referenced in the Collaboration by ClassifierRoles. These generated ClassifierRoles are then connected by the modeler via Message Interactions to the ClassifierRole representing the Activity’s performer BPRole, as well as other ClassifierRoles generated from BPRoles representing Artifacts “used” by Activities.

### *1.6.3 Business Event Model*

The EDOC Event Model provides the basis for specifying the exposure of some action of interest within a model element to some other part of the distributed application. The basic concepts are EventSources and EventSinks. An EventSource has rules which determine under what circumstances an action is exposed as a set of data attributes (an event) and transmitted to one or more EventSinks that are interested in it. EventSinks also have rules which determine which events of a particular type their attached model element wishes to receive.

In addition, the attributes of the event may be given values automatically by an implementation of the model by using expressions in the model which determine which values of which features of the Source model element should be assigned to which attributes of the event. Likewise, EventSinks can specify how the values of the attributes of an arriving event may be used to update the values of the Sink model element’s features. The aim of this model is to allow some kinds of event transmissions to be sufficiently well specified that all code that supports their implementation may be automatically generated. However, in some kinds of applications more specific implementation and mapping information is required before these decisions can be made, and in these cases programmers will be given hooks in mappings to technology that allow them to populate event contents using hand-written code.

## 1.7 *The Relationships between Models*

This specification provides an example of the application of the Model Driven Architecture that has come to prominence in the OMG. There are mappings defined which provide interaction points between different aspects of an EDOC application's model (often call viewpoints). These mappings provide a view of:

- the process model from the entity viewpoint (see Section B.2.6 “Mapping a Business Process as an Entity”).
- the entity model from the process viewpoint (see Section B.2.7 “Generation of an Activity from Features of an Entity”).
- the asynchronous aspects of the process model from the event viewpoint (see Section B.3 “Mapping Asynchronous DataGroups to Event Transceivers”).

There are also mappings defined between models at different levels of abstraction:

- the process model may be refined through a mapping to the EAI Profile of UML (see Section B.1 “EAI Mapping for the Business Process Model”).
- the process and event models may be realized as designs of CORBA interfaces and usage of CORBA service implementations (see Appendix C “Mapping to CORBA and Services”).

In addition proof of concept mappings, not documented here, have been carried out from entity and event models to implementations in Java and JMS, as well as from the process model to some proprietary workflow models.

### 1.7.1 *Entity Modeling*

We assume that the most popular and widely-used UML models will be used as a basis for Entity and Information modeling. The Class Model, Collaborations and Sequence Diagrams form the basis for modeling Objects and their relationships, as well as Interactions based on individual method calls and signal exchanges between them.

There are several ways in which entity models and process models rely on one another:

- From within the process model, BPRoles nominate Classifiers which type the objects that are used to enact Activities in a Business Process. A skeleton Collaboration is then produced with ClassifierRoles to represent the objects bound to BPRoles, and portals in and out of the process model. The EDOC designer will add Interactions and Messages indicating how the Objects indicated by the BPRoles in the process model are invoked to achieve the effect of the Activity.
- An existing Class may be mapped to a generated Activity within the process model. This allows the methods of the Class to be used as units of work in some larger process context.
- A Business Process will be represented by a Class that allows the process to be invoked using a set of operations or methods (each representing a variation on the initialization required to start a process). Instances of this generated Class may be called Business Process Objects.

### 1.7.2 Mapping Process aspects to Events

When mapping process models to implementation technologies, some platforms chosen will only support synchronous interactions, and therefore the asynchronous aspects of the process model must be mapped in some other way. One mechanism to do this is to map all asynchronous InputGroups and OutputGroups to EventSinks and EventSources attached to the Activities (or the Objects that bind to the BPRoles performing those Activities). The Event Model may then be mapped to a compatible implementation platform that can communicate the asynchronous flows via the messaging platform chosen.

### 1.7.3 Mapping to the EAI Profile

The UML Profile for EAI process has produced three first submissions: the largest of which (Concept Five Technologies, International Business Machines, Oracle, Rational Software and Unisys: ad/00-08-05 and ad/00-08-12) contains a model of event-driven application integration that includes all of the “wiring” infrastructure to provide a design for a messaging-oriented implementation of the EDOC process model, but does not include the coordination semantics that the process model captures. This work is the basis for a revised joint submission by all initial submitters. Appendix B provides two mappings to the EAI model, one of which assumes that the process coordination semantics are implemented in the systems being integrated, and the other uses EAI building blocks to simulate the semantics of the process model.

A mapping from the EDOC Event model to EAI is also straight-forward, but is not provided in this specification.

### 1.7.4 Mapping to CORBA and Services

Appendix C provides the non-normative mapping requested in the EDOC RFP. It shows how alternative IDL representations are possible for several EDOC model elements, and how different CORBA Services may be used to implement the same EDOC models.

## 1.8 Acknowledgments

This submission borrows many of the ideas developed as part of a workflow project carried out at the Department of Computer Science, University of Newcastle upon Tyne, UK. This workflow project was sponsored in part by Nortel Corporation.

Since the initial DSTC submission the IBM team, and in particular Marc-Thomas Schmidt, have provided valuable additional model concepts and alterations to the business process part of the first submission model.

Wojtek Kozaczynski of Rational Corporation was instrumental in bridging between concepts introduced by IBM and the initial DSTC EDOC submission.

Kathy Shan and her colleagues at Mincom provided us with the example in Appendix A as a validation of our model and notation.

The following members of the DSTC EDOC team were involved in producing this submission: Keith Duddy, Michael Lawley, Zoran Milosevic, Kerry Raymond, and Andrew Wood. They were supported by proof of concept prototyping by James Cole, Matt Davies, Anna Gerber and Jim Steel.

This work has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science & Resources).

## 2.1 Overview

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.

This model is organised with the *Task* abstract metaclass at its heart. A Task is a unit of work that has groups of inputs and groups of outputs. Some of these represent synchronizations with other tasks at the beginning and end of its life-cycle, while others represent asynchronous interactions during the lifetime of the Task. There are two concrete subtypes of Task: *Activity* and *CompoundTask*. CompoundTasks are the definitions of the way in which DataFlows connect the outputs of one Activity to the inputs of another. Activities are either performed by an entity that satisfies the

constraints of the *BPRoles* linked via the *performedBy* association, or they are performed according to the CompoundTask linked via the *definedBy* association. Many Activities may be *definedBy* the same CompoundTask.

A *BusinessProcess* is the encapsulation of a CompoundTask definition exposed to the enterprise system as a Classifier (usually a Class). Instances of this class will be objects that implement the CompoundTask that the BusinessProcess is *definedBy*. The Classifier support methods (and event receptions and emissions) representing the inputs and outputs of the CompoundTask that the BusinessProcess is definedBy. See Appendix B.

After examining the meta-model in Figure 2-1 on page 23, the attentive reader will realise that both an Activity and its associated CompoundTask contain InputGroups, OutputGroups, Inputs, and Outputs. The *DataMap* is used to link the Inputs and Outputs of an Activity with the Inputs and Outputs respectively, of its CompoundTask. The *DataMap*'s expression can be used to combine, split, and transform the incoming or outgoing data values. They allow us to de-couple the types and cardinality of a CompoundTask's Inputs and Outputs from the ones they are connected to in a particular Activity's context. This has obvious advantages when doing bottom-up modeling and dealing with legacy applications. For example, it allows off-the-shelf components, with interfaces that do not quite match, to be combined to implement the BusinessProcess.

Because we have decoupled Inputs and Outputs of Activities from the Inputs and Outputs of CompoundTasks, we are free to add arbitrary Inputs and Outputs to an Activity which are not mapped to the invoked CompoundTask. These can then be used as *control points*. That is, they can be used when one merely needs control flow for synchronisation purposes.

One can view the Inputs and Outputs of a CompoundTask as being like formal parameters while the Inputs and Outputs of Activities are like actual parameters.

We take a role-based approach to business entity modeling. That is, we enable a connection to be made between any entity model that is based on `Foundation::Core::Classifier` and our Business Process Model via a *BPRole*. This is motivated by the observation that business entities constitute a large class of things that may be modeled in many and varied ways using existing parts of UML (for example, a traditional Class model based on the Organizational Structure Facility<sup>1</sup> currently under standardization by the OMG). In the context of an EDOC business process, a *BPRole* represents a placeholder for the use of one (or more) of these entities as required to carry out an Activity. Valid entities are characterised by a `type` association and `factory` and `find` expressions. At runtime, the set of entities that satisfy both the `type` association and the `find` constraint will be determined and one of these entities will be chosen to be bound to the *BPRole*, otherwise the `factory` expression will be used to create an entity which is then bound to the *BPRole*.

While a *BPRole* is owned by a single CompoundTask, it may be associated with many Activities. This means that once the *BPRole* is bound to an Object, every Activity sees the same Object.

1.The RFPs OMG Document number is bom/99-11-03

We distinguish two kinds of BPRole use in our model: *performer roles*, those involved in performing an Activity via the `performedBy` association, and *artifact roles*, those used during the performance of the Activity via the `uses` association. Intuitively, the performer roles are the *subjects*, the artifact roles are the *objects*, and the Activity is the *verb*.

This rest of this section describes each element in the Business Process Model in detail. The model can be seen in Figure 2-1. Note that the containment associations are all derived associations from the `ElementOwnership` association inherited from `Namespace`.

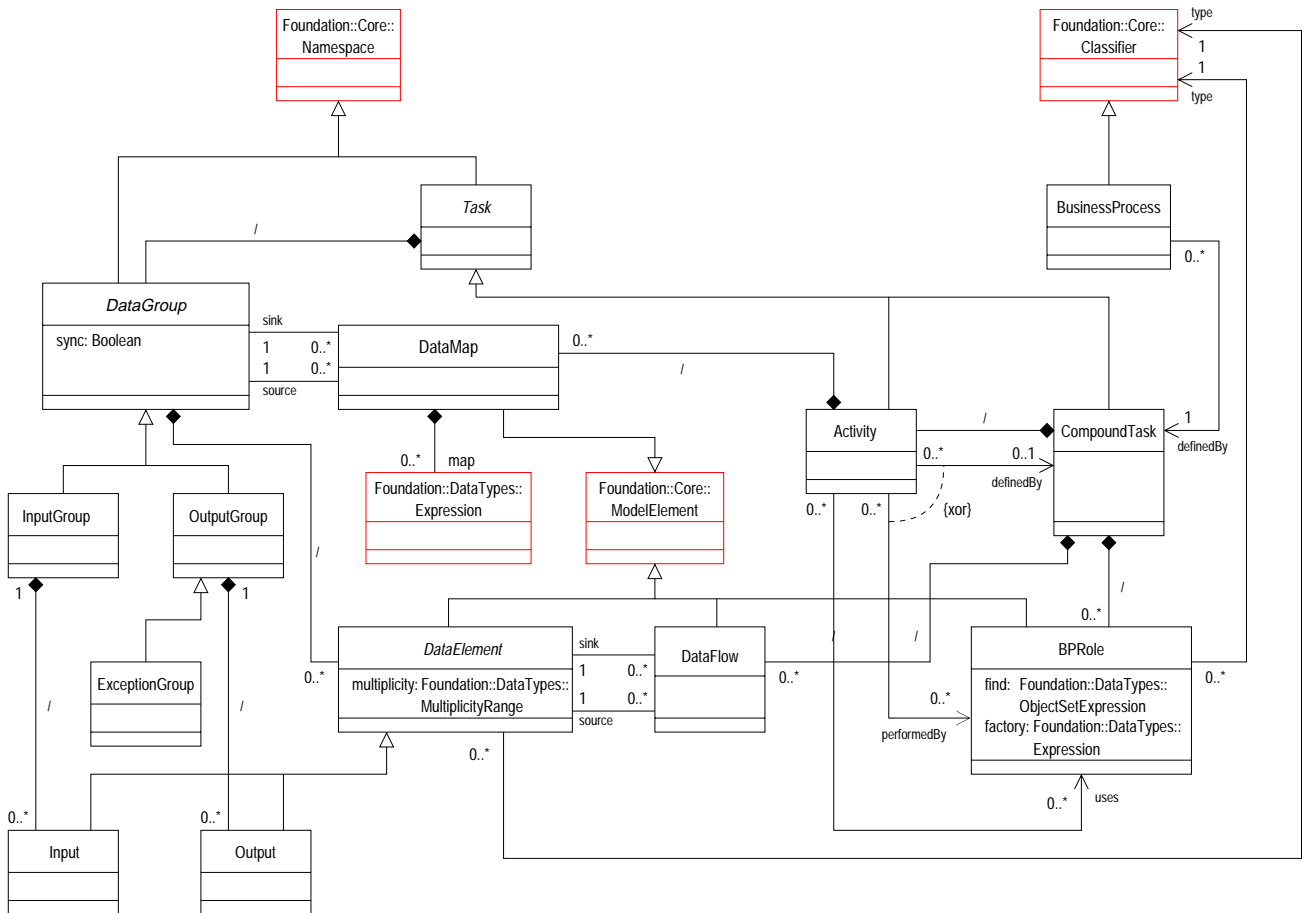


Figure 2-1 The Business Process Models.

## 2.2 Task (abstract)

### Purpose

`Task` represents a self-contained unit of work. It contains its `InputGroups`, `OutputGroups` and, indirectly, `Inputs` and `Outputs`.

## Inheritance

Task inherits from Foundation::Core::NameSpace, enabling it to contain other ModelElements. As a subclass of Namespace, Tasks have a name and the names of those ModelElements directly contained by a Task must be unique.

Task has two immediate (concrete) subtypes: Activity and CompoundTask

## Containment

The containment association is derived from the Namespace's ownedElement association.

All Tasks contain one or more DataGroups, one of which must be an implicit ExceptionGroup (see Section 2.9 "ExceptionGroup") that is named `system`, hereafter call the `system` ExceptionGroup. This can be represented explicitly in models for the purposes of handling system ExceptionGroups by connecting them via a DataFlow to a handler Activity. See Section 2.9 "ExceptionGroup" for further details.

Each InputGroup represents an alternative way to parameterise inputs to the Task. Each OutputGroup represents a possible output of the Task. They will have different semantics depending on whether the Task is an Activity or a CompoundTask, and on whether the DataGroup is synchronous or asynchronous. See the concrete metaclass definitions, Activity on page 35 and CompoundTask on page 36, for details.

## Runtime Semantics

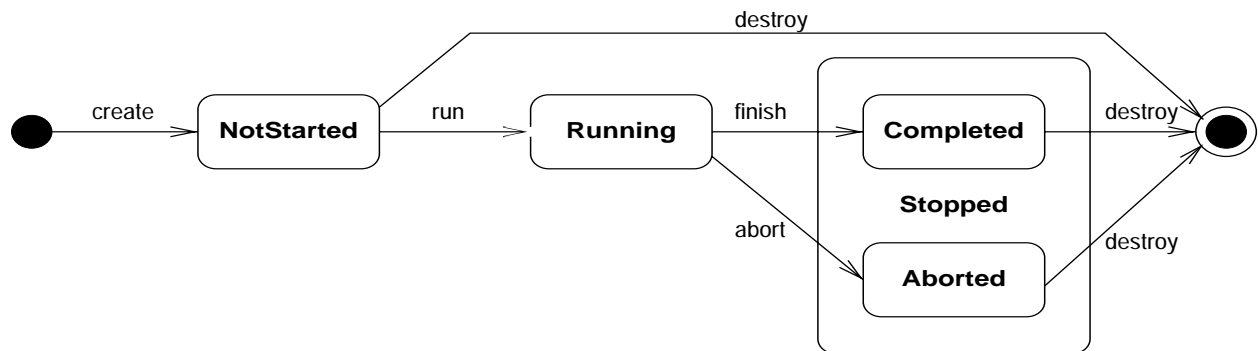


Figure 2-2 State machine for Task.

The concrete subtypes Activity and CompoundTask inherit the state machine of Task shown in Figure 2-2. However, the details of their behaviour in the various states are different and are given in Section 2.10 "Activity" and Section 2.11 "CompoundTask" below.

## 2.3 DataGroup (abstract)

### *Purpose*

*DataGroup* represents a set of related *DataElements* used to describe the inputs and outputs of a *Task*. They act as a form of correlator for *DataFlows*.

### *Attributes*

`sync` : Boolean

A value of TRUE indicates that this *DataGroup* represents either parameters that may be used to trigger a *Task* instance to enter the Running state, or results that are available when the *Task* instance enters the Stopped state.

A value of FALSE indicates that while the *Task* instance is in the Running state, the *DataGroup* may either asynchronously consume one or more sets of data, or asynchronously emit one or more sets of data.

### *Inheritance*

*DataGroup* inherits from *Foundation::Core::NameSpace*, enabling it to contain other model elements. Like *Task*, it also inherits the name constraints for its immediately contained *ModelElements*.

*InputGroup* and *OutputGroup* are (concrete) subtypes of *DataGroup*. A concrete *DataGroup* must be an *InputGroup* or an *OutputGroup*. A further derivation of *OutputGroup* is the *ExceptionGroup*, which indicates an extraordinary termination of a *Task* instance.

### *Containment*

A *DataGroup* is contained by a *Task*.

A *DataGroup* can contain zero or more *DataElements*. *InputGroups* contain only *Inputs*. *OutputGroups* contain only *Outputs*. This containment is re-stated explicitly via further derived associations in the Model diagram in Figure 2-1.

### *Run-Time Semantics*

A *DataGroup* instance is *satisfied* when **all** of its contained *DataElement* instances are satisfied (*AND semantics*), otherwise it is *unsatisfied*. If a *DataGroup* instance has no *DataElement* instances, it is (trivially) satisfied.

If a *DataGroup* instance is satisfied then it may be *enabled*. However, at most one synchronous *InputGroup* instance of a *Task* instance and one synchronous *OutputGroup* instance of a *Task* instance may be enabled and, once enabled, must remain in that state. An asynchronous *DataGroup* instance does not have these

constraints. It will enable its `DataElement` 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).

See the definitions of `InputGroup` on page 27 and `OutputGroup` on page 30 for more specific behavioural specifications.

## 2.4 `DataElement` (abstract)

### *Purpose*

`DataElement` represents data used in Task input/output.

### *Inheritance*

`DataElement` inherits from `Foundation::Core::ModelElement`.

`DataElement` has two subtypes, `Input` and `Output`. All `DataElements` must be either `Input` or `Output` (but may not be both).

### *Attributes*

`multiplicity` : `Foundation::Core::MultiplicityRange`

The `multiplicity` of a `DataElement` instance allows it to act as a collection of data values of the same type. The default multiplicity is `{1,1}` which represents a singleton collection.

### *Containment*

A `DataElement` is contained in a `DataGroup`. An `Input` is contained only by an `InputGroup`. An `Output` is contained only by an `OutputGroup`.

`DataElements` do not contain other model elements.

### *Associations*

A `DataElement` is optionally associated with a single `Foundation::Core::Classifier` by the *type* association. A `DataElement` that does not have an associated `type` can be thought of as a *'control point'*. That is, the values handled by these `DataElements` 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.

## *Run-Time Semantics*

A DataElement whose multiplicity has a lower bound of zero is an optional DataElement. Such a DataElement instance is always satisfied. If the multiplicity's lower bound, *lb*, is greater than zero then at least *lb* elements must be present in this DataElement instance before it is satisfied.

If more data values are added to a DataElement instance's collection than its upper bound allows, the collection will discard a value, and remain at the size of the upper bound. The choice of the value to be discarded is arbitrary.

Hence, a DataElement instance is satisfied when the number of data values it holds is in the range of its multiplicity.

A DataElement instance is enabled when its containing DataGroup is enabled.

A DataElement instance will reject an attempt to assign a value of an incompatible type.

When a DataElement instance is enabled it transmits its values using the associated DataFlow or DataMap instances as appropriate. If the DataElement instance is contained by an asynchronous DataGroup instance, its values are then discarded and the DataElement instance will become unsatisfied or satisfied according to its multiplicity bounds. After all DataElements have transmitted their values, the containing DataGroup instance will no longer be enabled and will become unsatisfied or satisfied based on the new states of its contained DataElements.

## 2.5 *InputGroup*

### *Purpose*

*InputGroup* models a set of data values required by a Task to do some work. When owned by an Activity, the *InputGroup* models the actual parameters to some behaviour of the Task. When owned by a CompoundTask the *InputGroup* models the formal parameters to some behaviour of the Task.

In the case of a synchronous *InputGroup* these values are a potential way to trigger the Task instance to enter the Running state.

### *Inheritance*

*InputGroup* inherits from *DataGroup*.

There are no subtypes of *InputGroup* identified in this specification.

### *Containment*

An *InputGroup* is contained in a Task.

An *InputGroup* can contain zero or more *Inputs*.

## *Run-Time Semantics*

An InputGroup instance is *satisfied* when all of its Input instances are satisfied, that is that they have received sufficient values (*AND semantics*), otherwise it is *unsatisfied*. If an InputGroup instance has no Input instances then it is (trivially) satisfied.

If an InputGroup instance is satisfied then it may be *enabled*. However, at most one synchronous InputGroup instance of a Task instance may be enabled. If more than one synchronous InputGroup instance of a Task instance is satisfied, then the choice among them of which one to enable is arbitrary (*Exclusive OR semantics*).

An InputGroup instance will enable its Input instances when it becomes enabled. If it is a synchronous InputGroup instance, then it triggers its containing Task instance to enter the `Running` state.

## 2.6 *Input*

### *Purpose*

*Input* models a collection of data values of a particular type consumed by a Task instance.

### *Inheritance*

Input inherits from DataElement.

### *Attributes*

Input has no direct attributes.

### *Containment*

An Input is contained in an InputGroup.

Inputs do not contain other model elements.

### *Associations*

An Input is optionally associated with a Foundation::Core::Classifier indicating its type.

Each Input owned by an InputGroup belonging to an Activity can be the sink of zero or more DataFlows, and may not be a source of any DataFlows (Figure 2-3).

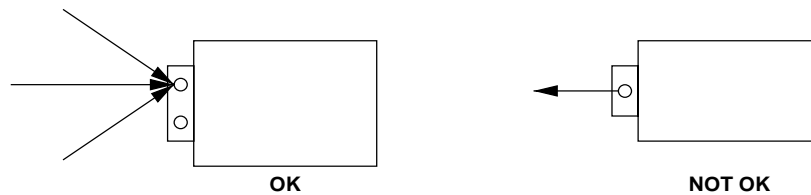


Figure 2-3 Inputs as sinks.

Each Input owned by an InputGroup belonging to a CompoundTask can be the source of zero or more DataFlows, and may not be a sink of any DataFlows (Figure 2-4).



Figure 2-4 Inputs as sources.

## Run-Time Semantics

An Input instance is *satisfied* when it has at least `multiplicity.lb` values, otherwise it is *unsatisfied*. It may not have more than `multiplicity.ub` values.

If an Input instance is the sink of more than one DataFlow, then data values for that Input 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 Input instance's collection remains at the size of the upper bound, and some arbitrary set of values are discarded.

If an Input instance is the target of more than one `map` Expression contained by a single DataMap instance, excess values will be similarly discarded.

When an Input instance is enabled and its containing Task instance is in the Running state, it transmits its values using the associated DataMap or all the associated DataFlows (*AND semantics*) as appropriate. If the Input 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`.

## 2.7 OutputGroup

### *Purpose*

*OutputGroup* represents a possible outcome of the execution of a Task; it provides data values associated with that outcome. In the case of a synchronous *OutputGroup* it also serves as an indication that the Task entered the *Stopped* state.

### *Inheritance*

*OutputGroup* inherits from *DataGroup*.

*OutputGroup* has a subtype *ExceptionGroup*, which signifies that its containing Task failed to perform the Task's function. Since *ExceptionGroups* are always synchronous, it also indicates the Task has entered the *Stopped* state.

### *Containment*

An *OutputGroup* is contained by a Task.

An *OutputGroup* contains zero or more *Outputs*.

### *Run-Time Semantics*

An *OutputGroup* instance is *satisfied* when all of its *Output* instances are satisfied, that is that they have received sufficient values (*AND semantics*), otherwise it is *unsatisfied*. If an *OutputGroup* instance has no *Output* instances then it is (trivially) satisfied.

If an *OutputGroup* instance is satisfied then it may be *enabled*. However, at most one synchronous *OutputGroup* instance of a Task instance may be enabled and only when the Task is in the *Stopped* state. If more than one synchronous *OutputGroup* instance of a Task instance is satisfied, then the choice among them of which to enable is arbitrary (*Exclusive OR semantics*).

An *OutputGroup* instance will enable its *Output* instances when it becomes enabled.

The enabling of a synchronous *OutputGroup* indicates that its Task has completed, that the enabled *OutputGroup* represents the outcome of the Task, and that the *Outputs* of that *OutputGroup* contain values appropriate to that outcome.

---

Note – The behaviour of asynchronous *OutputGroup* instances with multiple *Output* instances can be considered as correlation of asynchronously produced values from within a Task instance (either produced within the Process model by a *CompoundTask* instance, or within the Entity model by the object bound to the *performedBy* *BPRole* of this Activity instance).

---

## 2.8 Output

### *Purpose*

*Output* models a collection of data values of a particular type produced by a Task instance.

### *Inheritance*

Output inherits from DataElement.

### *Attributes*

Output has no direct attributes.

### *Containment*

An Output is contained in an OutputGroup.

Outputs do not contain other model elements.

### *Associations*

An Output has an optional `type` association with a `Foundation::Core::Classifier`.

Each Output owned by an OutputGroup belonging to a CompoundTask can be the sink of zero or more DataFlows (Figure 2-5).



Figure 2-5 Outputs as sinks.

Each Output owned by an OutputGroup belonging to an Activity can be the source of one or more DataFlows (Figure 2-6).

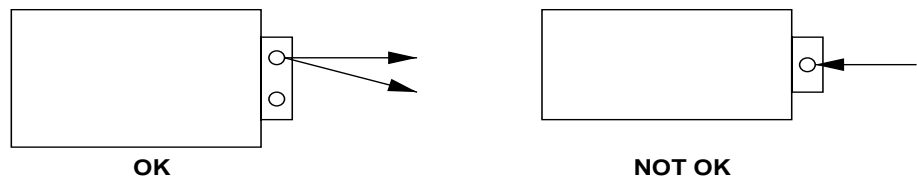


Figure 2-6 Outputs as sources.

## *Run-Time Semantics*

An Output instance is *satisfied* when it has at least `multiplicity.lb` values, otherwise it is *unsatisfied*. It may not have more than `multiplicity.ub` values.

If an Output instance is the sink of more than one DataFlow, then data values for that Output 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 Output instance's collection remains at the size of the upper bound, and some arbitrary set of values are discarded.

If an Output instance is the target of more than one map Expression contained by a single DataMap instance, excess values will be similarly discarded.

If an Output instance contained by a synchronous OutputGroup instance is enabled and its containing Task instance is in the `Stopped` state, then it transmits its values using the associated DataMap or all the associated DataFlows (*AND semantics*) as appropriate.

If an Output instance contained by an asynchronous OutputGroup instance is enabled and its containing Task instance is in the `Running` state, then it transmits its values using the associated DataMap or all the associated DataFlows (*AND semantics*) as appropriate. It then discards its values and resets its state to `unsatisfied` or `satisfied` according to its `multiplicity`.

## 2.9 *ExceptionGroup*

### *Purpose*

*ExceptionGroup* represents the outcome of a Task that failed to complete its function. If desired, an Activity's *ExceptionGroup* can be *handled* either by an exception handler (an Activity) or by an *ExceptionGroup* of the containing *CompoundTask*. If, at runtime, an Activity instance's *ExceptionGroup* is not handled and the *ExceptionGroup* is enabled, then it will be *propagated*. That is, the containing *CompoundTask* instance's `system` *ExceptionGroup* will be enabled (which consequently causes the *CompoundTask* to abort its contained Activity instances and terminate in the `Aborted` state).

### *Inheritance*

*ExceptionGroup* inherits from *OutputGroup*.

There are no subtypes of *ExceptionGroup* identified in this Specification.

### *Attributes*

*ExceptionGroup* has no direct attributes. The value of the attribute `sync` inherited from *DataGroup* must be *TRUE*.

## Containment

Like OutputGroups, ExceptionGroups are contained by Tasks. A Task can contain one or more ExceptionGroups.

Like OutputGroups, ExceptionGroups can contain zero or more Outputs.

## Associations

Like any OutputGroup owned by an Activity, an Outputs contained by an ExceptionGroup can be the source of DataFlows. If an Activity's ExceptionGroup contains an Output that is the source of a DataFlow, then the ExceptionGroup is considered to be *handled*, otherwise it is considered to be *unhandled*.

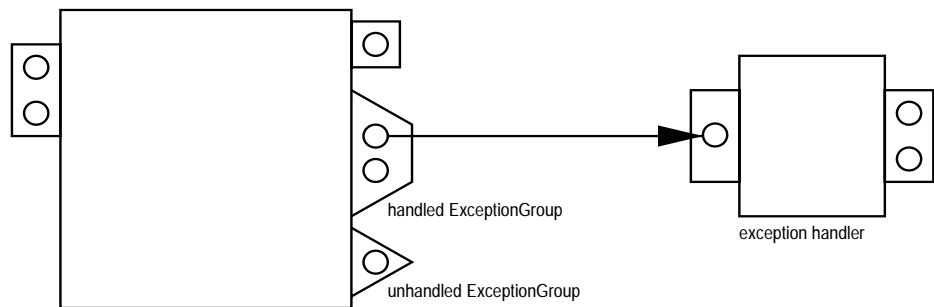


Figure 2-7 Handling an ExceptionGroup.

It is the modeler's responsibility to ensure that the sink InputGroup or ExceptionGroup of a DataFlow sourced by an ExceptionGroup is capable of responding to the exception in a timely manner or at all. Figure 2-7 illustrates an exception handler that handles an ExceptionGroup, while Figure 2-8 illustrates a poorly designed exception handler since it will only ever be able to respond to a single ExceptionGroup.

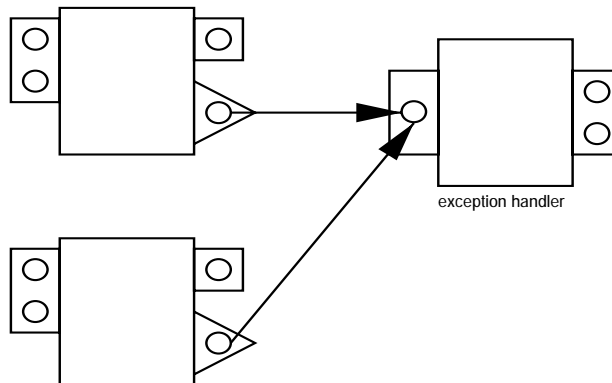


Figure 2-8 Poorly designed ExceptionGroup handling.

A better alternative to Figure 2-8 is shown in Figure 2-9.

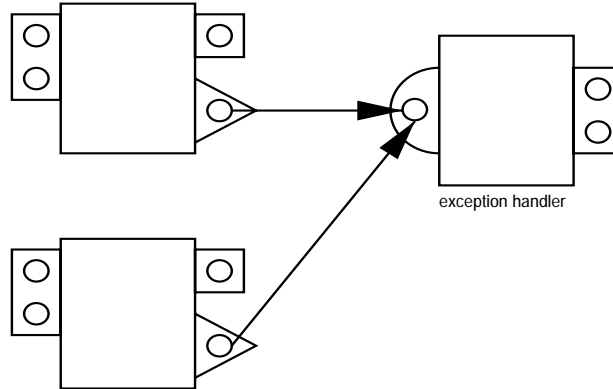


Figure 2-9 Handling potentially many exception with an asynchronous InputGroup.

Figure 2-10 illustrates how more than one ExceptionGroup of contained Activities can be propagated to the same ExceptionGroup of the CompoundTask.

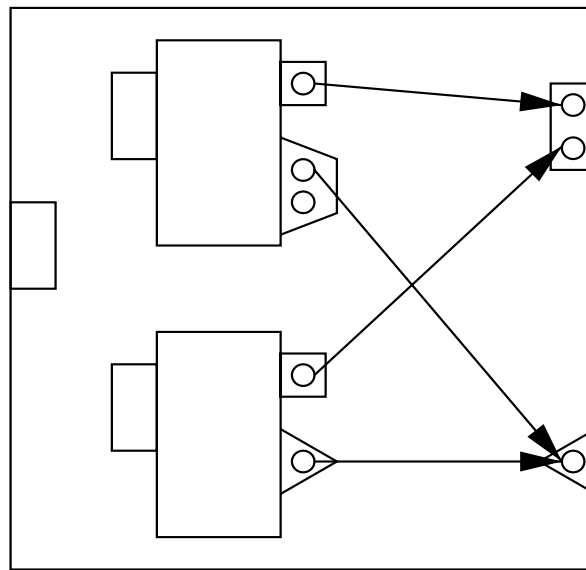


Figure 2-10 Propagating an ExceptionGroup.

### *Run-Time Semantics*

When a handled ExceptionGroup is enabled, it enables the DataFlows for which its contained Outputs are sources.

When an unhandled ExceptionGroup (of an Activity) is enabled, the system ExceptionGroup of the containing CompoundTask is enabled and, consequently, all running Activities contained by that CompoundTask will be terminated. The CompoundTask will then satisfy its completion criteria.

## 2.10 Activity

### *Purpose*

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

- The creation of an instance of a CompoundTask, referred to via the Activity's *definedBy* association.
- The execution of some feature of an Object bound to a BPRole instance referred to via the Activity's *performedBy* association. (See "BPRole" on page 41.)

The InputGroups contained by an Activity represent the alternative means by which the Activity may supply data to these mechanisms to initiate some action.

Synchronous InputGroup instances 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 InputGroups (synchronous or asynchronous) from its containing CompoundTask.

If no Synchronous InputGroups are present, then the Activity will be initialised as part of the initialization of its containing CompoundTask. This will allow it to receive asynchronous input as soon as they propagate into the containing CompoundTask.

When an Activity is performedBy a PerformerRole which has not yet been bound, the PerformerRole will be bound to an appropriate Object during the initialization of the Activity. The binding for the Role will last for the duration of the life time of the CompoundTask, but the Object it binds to may exist before the binding is created, and may live longer than the binding.

Asynchronous InputGroups 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 InputGroups is enabled) all data values that arrive at an asynchronous InputGroup will be kept in that InputGroup's Inputs only up to their 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 and transmitted via the Activity's DataMaps.

---

Note – This behaviour trades off the resource savings of keeping asynchronous values only up to and including the slots defined by an InputGroup and its 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' InputGroups 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 InputGroups will be queued, never to be consumed by that Activity.

---

## *Inheritance*

Activity inherits from Task.

## *Associations*

An Activity is either associated with a CompoundTask via the `definedBy` association or it is associated with one or more BPRoles via the `performedBy` association.

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

An Activity may also be associated via the `uses` association to one or more BPRoles. These BPRoles will be bound to Objects at run time as the Activity enters the `Running` state.

## *Containment*

Activities can contain zero or more DataMaps.

## *Run-Time Semantics*

Figure 2-2 shows the state machine for an Activity instance. After the Activity instance is created it enters the `NotStarted` state. Once one of its synchronous InputGroup instances is enabled (or it has no synchronous InputGroup instances), BPRole binding is performed (as specified for BPRole on page 34), a CompoundTask instance is created if the `definedBy` association exists, and the Activity instance enters the `Running` state. While in the `Running` state, values from enabled Input instances may be consumed.

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.

## 2.11 CompoundTask

### *Purpose*

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 BusinessProcess.

### *Inheritance*

CompoundTask inherits from Task.

## Containment

CompoundTask can contain zero or more Activities.

A CompoundTask can contain zero or more DataFlows that connect its Inputs and Outputs and the Inputs and Outputs of its contained Activities.

A CompoundTask contains zero or more InputGroups and one or more OutputGroups. One of these OutputGroups is an implicit ExceptionGroup named `system`.

## Run-Time Semantics

Figure 2-2 shows the state machine for a CompoundTask instance. When a CompoundTask instance is created, it begins by creating instances of each of its immediately contained Activities (and DataGroups, DataElements, and DataFlows). If the CompoundTask instance has no synchronous InputGroups, or it has a synchronous InputGroup that is enabled, then it enters the `Running` state. While in the `Running` state, values from enabled Input instances may be consumed.

A CompoundTask instance enters the `Completed` state when none of its contained Activity instances that have synchronous OutputGroup instances (that are not also ExceptionGroup instances) 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 OutputGroups) are running. This results in a *quiescent* model for completion.

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

If a CompoundTask instance enters the `Completed` state, then a `satisfied` synchronous OutputGroup is enabled. If there is more than one `satisfied` synchronous OutputGroup, then the choice of which one to enable is arbitrary. If there is no synchronous OutputGroup that is `satisfied`, then the CompoundTask instance's `system` ExceptionGroup is enabled.

If an Activity instance contained by a CompoundTask enters the `Completed` state with an ExceptionGroup enabled and this ExceptionGroup is unhandled (see Section 2.9 "ExceptionGroup" for the definition of handled and unhandled ExceptionGroups), then the containing CompoundTask instance's `system` ExceptionGroup is enabled.

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

## 2.12 BusinessProcess

### *Purpose*

*BusinessProcess* establishes a context within which a set of business actions taking place in a prescribed manner are coordinated to achieve some enterprise objective.

### *Inheritance*

*BusinessProcess* inherits from *Foundation::Core::Classifier*. This enables a *BusinessProcess* to be a target of role binding. A standard mapping between the *InputGroups* and *OutputGroups* of the *CompoundTask* that defines the *BusinessProcess* and the features of the *Classifier*. (Methods to represent the synchronous *InputGroups*, *Receptions* to represent the asynchronous *InputGroups*, and *Associations* to “call-back” classes to represent the *OutputGroups*.) See Appendix B.

### *Associations*

A *BusinessProcess* is enacted according to the *CompoundTask* referenced by the *definedBy* association.

### *Run-Time Semantics*

When a *BusinessProcess* instance is created, an instance of the *CompoundTask* referenced by the *definedBy* association is created. The features of the *BusinessProcess* instance may then be used to provide values to the *CompoundTask* instance’s *InputGroups*, and retrieve values from its *OutputGroups*.

## 2.13 DataFlow

### *Purpose*

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 *DataElements*. In the case that a *DataFlow* connects two *DataElements* in synchronous *DataGroups*, the implication is that the *Activities* occur in strict temporal sequence.

### *Containment*

*DataFlows* are contained by a *CompoundTask*; a *CompoundTask* can contain zero or more *DataFlows*.

*DataFlows* do not contain other model elements.

## Associations

A DataElement is a source of a DataFlow. A DataFlow has exactly one source DataElement, but a DataElement can be the source of zero or more DataFlows.

A DataElement is a sink of a DataFlow. A DataFlow has exactly one sink DataElement, but a DataElement can be the sink of zero or more DataFlows.

The DataElement that is the source of a DataFlow must be contained (indirectly) by the same CompoundTask as the DataFlow, and must be either:

- an Input of an InputGroup of the DataFlow's containing CompoundTask; or
- an Output of an OutputGroup of an Activity directly contained by the DataFlow's containing CompoundTask.

A DataElement that is the sink of a DataFlow must be contained (indirectly) by the same CompoundTask as the DataFlow, and must be either:

- an Output of an OutputGroup of the DataFlow's containing CompoundTask; or
- an Input of an InputGroup of an Activity directly contained by the DataFlow's containing CompoundTask.

The well-formed-ness rules above can be visualised as “DataFlows cannot cross the boundaries of CompoundTasks”. Figure 2-11 shows three illegal DataFlows (see how the illegal DataFlows cross Task boundaries).

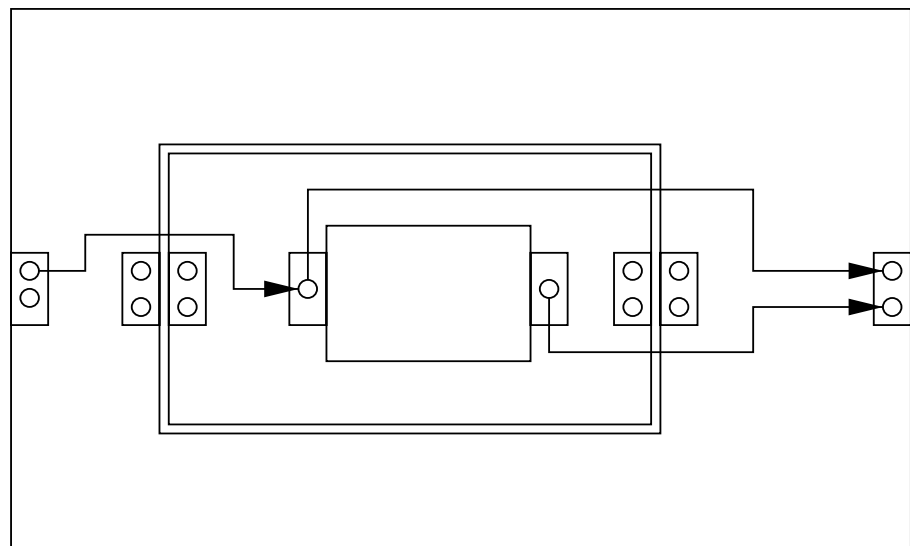


Figure 2-11 Illegal DataFlows crossing Task boundaries.

The type of the source DataElement of a DataFlow must be the same as (or coercible to) the type of the sink DataElement of a DataFlow. Coercible includes converting a value of type `T` to a member of type `collection<T>` and vice versa.

DataFlows between synchronous DataGroups within a CompoundTask should be acyclic; that is, things cannot happen in a circular order. (However, see Business Process Patterns on page 61 for how to specify processes involving looping.)

### *Run-Time 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` DataElement to the `sink` DataElement. The `sink` DataElement may then discard values as necessary if its `multiplicity` upper bound is reached.

## 2.14 DataMap

### *Purpose*

A DataMap is used to link the Inputs of an Activity's InputGroup with the Inputs in its `definedBy` CompoundTask's InputGroups and to link the Outputs in an Activity's OutputGroups with the Outputs in its `definedBy` CompoundTask's OutputGroups.

### *Containment*

A DataMap is contained by an Activity. An Activity contains zero or more DataMaps.

A DataMap contains zero or more `map` Expressions. The `body` attribute of each Expression should consist of an assignment statement that indicates how a DataElement of the sink DataGroup gets its value(s) from the DataElements of the source DataGroup. If there are no `map` Expressions, then a one-to-one assignment of DataElements is assumed and the set of locally scoped DataElement names of the sink DataGroup must be a subset of the set of locally scoped DataElement names of the source DataGroup.

### *Associations*

A DataGroup is the source of a DataMap. A DataMap has exactly one source DataGroup, and a DataGroup can be the source of zero or many DataMaps, but only one per Activity. That is, every InputGroup of an Activity must be the source of exactly one DataMap, and every OutputGroup of the CompoundTask that an Activity is `definedBy` must be the source of exactly one DataMap owned by that Activity.

A DataGroup is the sink of a DataMap. A DataMap has exactly one sink DataGroup, but a DataGroup can be the sink of zero or more DataMaps.

The values of the `sync` attribute for the source and sink DataGroups of a DataMap must be the same. That is, they must either both be synchronous or both be asynchronous. Additionally, after the DataMap has been evaluated the sink DataGroup must be satisfied. Depending on the language used for the DataMap's maps, tools may be able to check and/or enforce this.

## *Run-Time Semantics*

If a DataMap instance's Activity is in the `Running` state and the DataMap instance's source DataGroup instance is enabled, then DataMap's map expressions are evaluated to transform the values of the source DataElements into values for the sink DataElements.

## 2.15 BPRole

### *Purpose*

*BPRole* defines a placeholder for entities that perform an Activity or are used in the performing of an Activity. It defines a placeholder for behaviour in a context. The context is a CompoundTask (established with some objective in mind) and the behaviour of the BPRole becomes part of the behaviour of the CompoundTask as a whole. A BPRole is defined by its `type`.

### *Inheritance*

BPRole inherits from `Foundation::Core::ModelElement`.

### *Containment*

A BPRole is contained by exactly one CompoundTask. The CompoundTask provides the context in which the role-entity binding exists. While the lifetime of the entity is independent of the CompoundTask instance and any bound BPRole instance, the lifetime of a BPRole instance and its binding cannot exceed that of the BPRole instance's containing CompoundTask instance.

### *Associations*

A BPRole can be associated with an Activity via the `performedBy` association. This can be read as: An Activity is performed by a BPRole.

A BPRole may also be associated with an Activity via the `uses` association. This can be read as: An Activity uses a BPRole.

A BPRole `type` is described through the association of a BPRole model element with a class whose instances are of the specified `type`.

## *Attributes*

`find` : `Foundation::DataTypes::ObjectSetExpression`

The `find` attribute of a `BPRole` further constrains the set of entities that may be bound to the `BPRole` at runtime. It may refer to the values of `InputGroups` of the associated `Activities`. This allows parameterized late-binding of resources.

`factory` : `Foundation::DataTypes::Expression`

The `factory` attribute of a `BPRole` can be used to generate entities to be bound to the `BPRole` in the case that the `find` expression is empty or evaluation of `find` produces an empty set.

## *Run-Time Semantics*

When an `Activity` is enabled, binding of any associated unbound `BPRole` instances ensues based on the values of the `find` and `factory` Expressions. Note, some `BPRole` 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 `find` and `factory` 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 `BPRole` instance begins by determining the *candidate entities*. These are the set of entities with a compatible `type` and that further satisfy the `find` constraint. The `find` constraint may refer to the values of attributes of the `InputGroups` of any of its associated `Activities`. It is incumbent on the modeler to ensure that the `find` constraint is well-formed in the face of attributes that may not yet have values.

If there are no candidate entities, and the `factory` Expression is non-empty, it will be used to generate a new candidate entity (or entities if the Expression returns multiple instances).

One of the candidate entities will then be bound to the `BPRole` instance. If there are no candidate entities, the containing `CompoundTask` instance will have its `systemExceptionGroup` enabled.

We note that something akin to the OMG Trader service can be used for this binding process as discussed in the non-normative mapping (see Appendix C.2.6). Also, the bound entity may be a proxy for a person such as a worklist in a workflow execution environment.

2.16 Runtime Semantics Summary

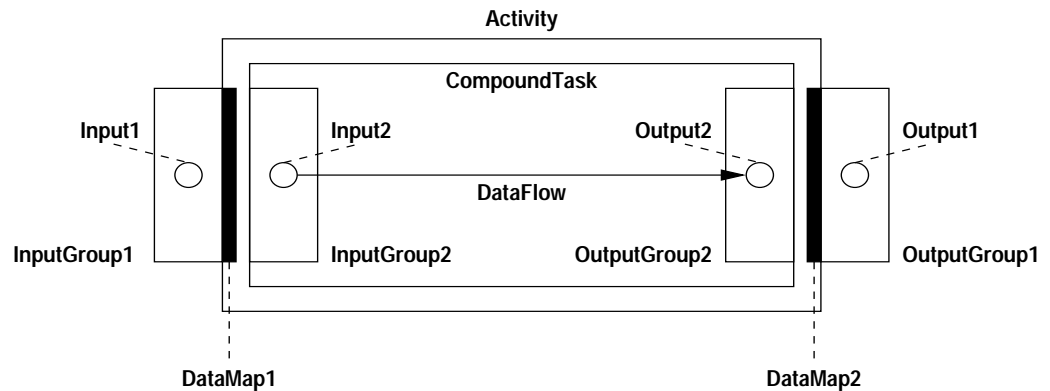


Figure 2-12 Example Activity and the CompoundTask it is defined by.

While clearly not illustrating all features of the Business Process Model, the example Activity and (trivial) CompoundTask shown in Figure 2-12 covers the essential elements and, by examining the provided execution trace, the reader should be able to grasp the details of the runtime semantics sufficiently to fill in the gaps of missing elements.

The expected execution trace is as follows:

- Input1, InputGroup1, Output1, and OutputGroup1 are unsatisfied.
- Activity is NotStarted.
- A value is assigned to Input1 satisfying its lower bound multiplicity constraint.
- Input1 becomes satisfied, followed by InputGroup1.
- InputGroup1 is then enabled which enables Input1 and DataMap1.
- Activity enters the Running state and creates the CompoundTask instance (which creates InputGroup2, Input2, OutputGroup2, Output2, and DataFlow).
- Being enabled and with Activity in the Running state, DataMap1 flows values from Input1 to Input2.
- Input2 becomes satisfied which satisfies InputGroup2.
- InputGroup2 is then enabled which enables Input2 and DataFlow.
- DataFlow flows values from Input2 to Output2.
- Output2 becomes satisfied, followed by OutputGroup2.
- There are no Activity instances in CompoundTask with synchronous OutputGroups that are not also ExceptionGroups and that are in the Running state so CompoundTask now enters the Completed state.
- OutputGroup2 can then enter the enabled state which enables Output2.
- DataMap2 is then enabled and flows values from Output2 to Output1.
- Output1 becomes satisfied, followed by OutputGroup1.
- Activity enters the Completed state allowing OutputGroup1 to be enabled followed by Output1. Output1's values would now be available for flowing by a DataFlow having Output1 as its source.



### *3.1 Introduction & Overview*

The Business Event Model shown in Figure 3-1 on page 46 allows sources and sinks of asynchronous multicast events to be attached to various EDOC Model Elements to allow them to expose their actions or state changes to other parts of the application. Restrictions can be placed on how widely the events are broadcast, ultimately allowing point to point events in some cases. It is left to mappings of this Model to determine the transmission mechanism for events.

The notion of an “event” instance is defined via the `type` association with a UML Classifier allowing it to be mapped to a number of different implementation technologies. A business event explicitly exposes an action that has a significant business semantics with respect to the system being modeled or its environment.

In order for EventSources and EventSinks to operate without any hand-written code, they have an `exposureRule` to determine when they should either emit or consume events. In addition, the population of the event instance property values to be emitted from an EventSource can be done automatically using the associated `map Expressions`; mapping from ModelElements in the source that represent values to the properties/attributes of the event. At the EventSink the use of the event’s properties can also be defined in terms of `map Expressions` from named properties of the event consumed into ModelElements contained or referenced by the sink ModelElement which represent values of the same type (or some type for which a standard coercion of the value is defined). However, regardless of the presence of maps, the EventSources and EventSinks will be exposed to programmers allowing them to build events to be emitted, and use events that are consumed, using their own code.

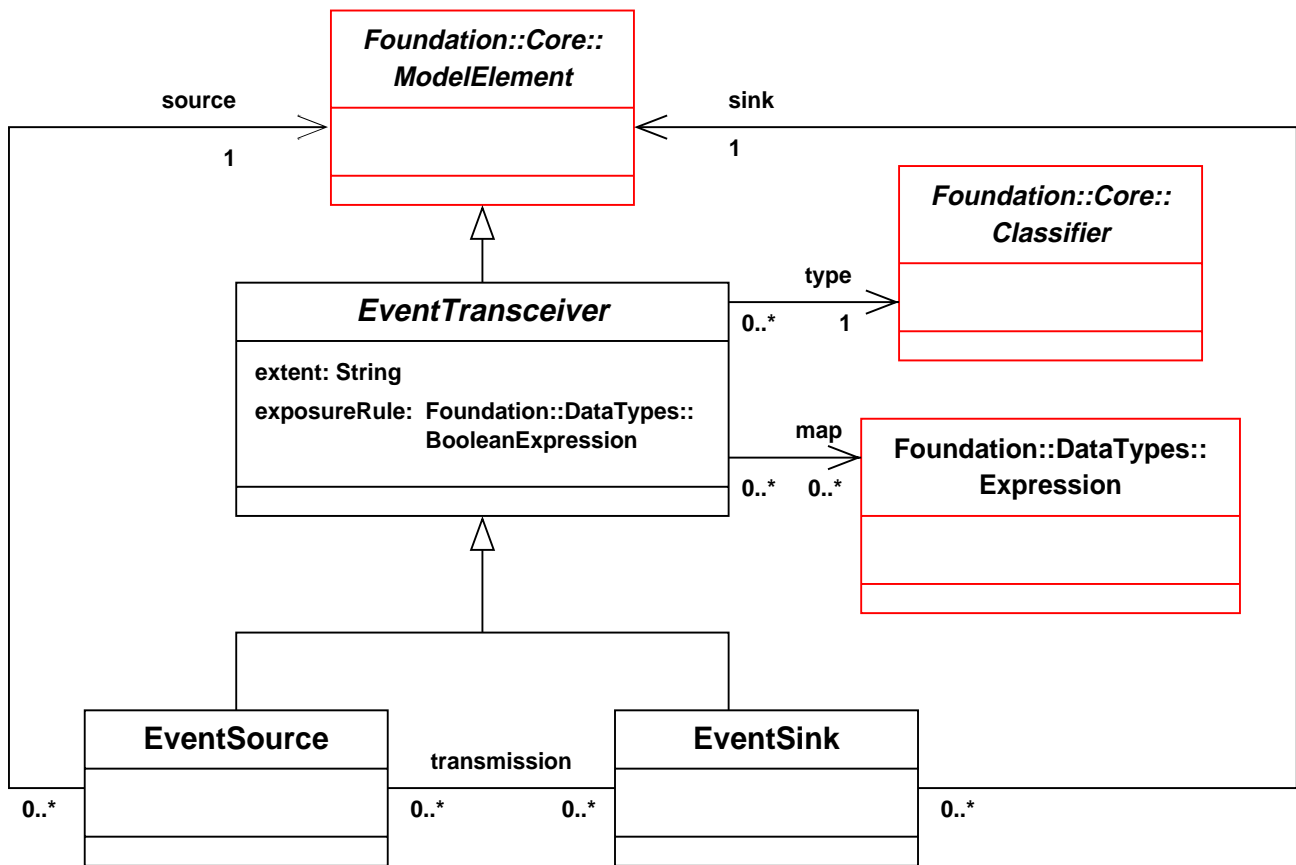


Figure 3-1 The Event Model.

## 3.2 EventTransceiver (abstract)

### Purpose

An EventTransceiver represents the external exposure of some action within an EDOC ModelElement to the rest of a Business application, or the consumption of an event representing that action. This abstract class represents the features common to both sources and sinks of events.

### Inheritance

EventTransceiver inherits from Foundation::Core::Model Element.

## Attributes

`exposureRule`: The conditions under which a business event is emitted or consumed.

The `exposureRule` may be evaluated in the context of an `EventSource`. In this case, the terms of the expression can be the attributes of (and other elements contained by or associated with) the `source` `ModelElement`. At a `source` there are a number of standard names that can be used as `exposureRule` terms:

- `"action"`: a String which indicates the name of an action occurring in the `source` `ModelElement`. Some standard action strings are given for particular `ModelElements` in Section 3.5.
- `"transition"`: a String which indicates that a named transition was taken in a state machine implemented in the `source` `ModelElement`. Unnamed transitions will be represented as the empty string.
- `"from_state"`: a String which indicates that a transition in a state machine implemented in the `source` `ModelElement` was made from the named state to another state.
- `"to_state"`: a String which indicates that a transition in a state machine implemented in the `source` `ModelElement` was made to the named state from another state.

Additional rule terms may be available depending on which `source` `ModelElement` the `EventSource` is attached to. See Section 3.5 for details.

Alternatively, the `exposureRule` is evaluated over the properties of an event arriving at an `EventSink` and the attributes of the `sink` `ModelElement`. The range of named elements that can be terms is dependent on the expression language. For example, the Trader constraint language only allows simple names, whereas OCL allows containment and other associations to be traversed.

`extent`: A name for the extent in which the event is to be broadcast or listened to. Three reserved names are available:

- `"global"`, indicating that the event may be sent outside the scope of the EDOC application being Modeled, or consumed from outside that scope;
- `"application"`, indicating that this event may only be broadcast or consumed by elements in the scope of the EDOC application being modeled;
- `"direct_only"`, indicating that events may only be sent between sources and sinks participating in the `transmission` association.

This attribute also permits naming schemes for particular roles or entities in the model to which events may be sent directly. It also allows for descriptions of topics or channels that will be refined when a design for event-based configuration is provided and a mapping to technology chosen.

## *Associations*

`type`: The type of the business event to be emitted or consumed.

`map`: The way in which values of an event's attributes are assigned from an emitting ModelElement's attributes, or assigned to a consuming ModelElement's attributes.

## *Run-Time Semantics*

See the derived concrete elements EventSource and EventSink.

## *Tool Considerations*

Tools may restrict the kinds of classifiers that are referenced by the `type` association to be value-types.

Tools may provide naming schemes for `extent` that indicate direct transmission of events to particular entities or groups of entities.

## 3.3 EventSource

### *Purpose*

An EventSource is used by a modeler to expose the actions of some EDOC ModelElement to the EDOC application as an event. This is done by choosing an appropriate type for the event, and providing an `exposureRule` which specifies under what conditions the event will be emitted. Finally, the EventSource may have a set of `maps` from ModelElements associated with the `source` ModelElement to event properties/attributes. Values referred to by any given mapping must have the same type, or have types that may be appropriately coerced.

The event is made available to programmers implementing the `source` ModelElement regardless of whether `maps` are supplied for automated assignment of values.

### *Inheritance*

EventSource inherits from EventTransceiver.

### *Attributes*

`exposureRule`: The conditions under which an event is emitted. The `exposureRule` may refer to the names of attributes of the `source` ModelElement, or to certain elements contained in or associated with that ModelElement. The range of elements accessible varies depending on the expressive power of the rule language, and on additional relationships defined in this document on an element by element basis.

`map`: The way in which the values in the event's attributes are assigned from the `source` `ModelElement`'s attributes, or from other elements contained in or associated with that `ModelElement`. The set of elements that can be the source of values to be assigned to an event is defined on an element by element basis.

## *Associations*

`source`: The `ModelElement` which performs the action to be exposed as an event.

`transmission`: Asserts that events emitted from an `EventSource` must be distributed in such a way as to reach the `EventSink` at the other end of the association.

## *Run-Time Semantics*

Conceptually, for every action that the `source` `ModelElement` takes, the `exposureRule` is evaluated and, if true, an event is constructed in accordance with the `maps`, then emitted.

(At least) the following `ModelElements` perform actions and can therefore be producers (sources) of business events:

- `Activity`
- `InputGroup`, `OutputGroup`, `ExceptionGroup`
- `Input`, `Output`
- `BPRole`

## 3.4 *EventSink*

### *Purpose*

An `EventSink` is used by a modeler to consume events representing the actions of some other `ModelElement`. This is done by choosing an appropriate `type` for the events to be consumed, and providing an `exposureRule` which specifies under what conditions the event will be consumed. Finally, the `EventSink` may have a set of `maps` from event properties/attributes to `ModelElements` associated with the `sink` `ModelElement`. Values referred to by any given mapping must have the same type, or have types that may be appropriately coerced.

The event is made available to programmers implementing the `sink` `ModelElement` regardless of whether `maps` are supplied for automated assignment of values.

### *Inheritance*

`EventSink` inherits from `EventTransceiver`.

## *Attributes*

`exposureRule`: The conditions under which an event is consumed. The `exposureRule` may refer to the names of event properties, attributes of the `sink` `ModelElement`, and to the names of certain other elements contained in or associated with that element. The range of the elements that can be terms in the rule depends on the expressive power of the rule language, and on additional relationships defined in this document on an element by element basis.

`map`: The way in which the values in the event's attributes are assigned to the `sink` `ModelElement`'s attributes, or to other elements contained in or associated with that `ModelElement`. The set of elements that can have values assigned to them is defined on an element by element basis.

## *Associations*

`sink`: the `ModelElement` which will use the event being consumed.

`transmission`: Asserts that events emitted from an `EventSource` at the other end of the association must be distributed in such a way as to reach this `EventSink`.

## *Run-Time Semantics*

A subscription is made to broadcast events of associated `type` and, for every delivered event for which the `exposureRule` is evaluates to true, values from the event are assigned to `sink`'s attributes and other contained or associated elements according to the `maps`. See Section 3.5 for details of which elements may be assigned to.

(At least) the following `Model Elements` can be consumers (sinks) of business events:

- Activity
- InputGroup, OutputGroup, ExceptionGroup
- Input, Output
- BPRole

## 3.5 *Actions at Sources*

This section nominates which actions conceptually cause the evaluation of the `exposureRule` in an `EventSource`. (Mappings to implementations will probably optimize the number of actual evaluations made at run time.)

### 3.5.1 *Model Element*

The event transmission architecture described in the Model allows for any UML `ModelElement` to have an `EventSource` or `EventSink` attached to it. Given the visibility rules of UML models the set of terms in the `exposureRule` and `map` Expressions of these sources and sinks can be determined. Then depending on the mapping of the `ModelElements` in question, code can be generated to use the Features of the `ModelElements` to populate events and transmit them to their recipients.

For example, a Class could have an EventSink attached which relied on the values of the Attributes of this Class in its `exposureRule` and whose map Expressions nominated other Attributes in Classifiers accessible via Associations in which instances of the Class participated.

Furthermore, any ModelElement whose behaviour was specified by a StateMachine could have an attached EventSource which emitted an event when certain states were reached, or transitions made.

## 3.5.2 Activity

An Activity has an associated finite state machine (see Figure 2-1). An action of an Activity corresponds to making a transition. Along with the Activity's attributes (see Task and its subclasses in the Business Process Model, Figure 2-1), the pseudo-attributes `'transition'`, `'from_state'`, and `'to_state'` (all of type String) are available for use in the `exposureRule` and the map Expressions.

An Activity also contains a number of InputGroups and OutputGroups, and the Inputs and Outputs of these elements can be named by using the form `"input-group-name::input-name"`. These names can be used in rule expressions that expect the type of the Input or Output, or may be compared with the special value NULL. In addition, the names of Inputs and Outputs may be used in map Expressions and, when nominated at a `sink`, the assignment of a value into the Input or Output is equivalent to such a value arriving from a DataFlow.

## 3.5.3 DataGroup

A DataGroup has two actions: `'satisfy'` (once all its inputs have been satisfied), and `'enable'` (when it is selected to start an Activity or as the result of an Activity completing). Along with the DataGroup's attributes (see DataGroup and its subclasses in the Business Process Model, Figure 2-1) including the values of each of the contained DataElements, the attribute `'action'` is available for use in the `exposureRule` and the map Expressions. The attribute `'action'` will have the value `'satisfy'` or `'enable'`.

The names of the Inputs or Outputs within this DataGroup may be used in `exposureRules` and map Expressions and, as in the case of Activity, the assignment of values to one of these is equivalent to their arrival via a DataFlow.

## 3.5.4 DataElement

A DataElement's actions are to be assigned a value, to become satisfied and to be enabled. The DataElement's attributes (see DataElement and its subclasses in the Business Process Model, Figure 2-1) are available for use in the `exposureRule` and the map Expressions. The attribute `'action'` may have the value `'assign'`, `'satisfy'`, or `'enable'`.

## *3.5.5 BPRole*

This is a special case of the ModelElement description in section 3.5.1. Roles have two states: bound and unbound. The attribute 'action' may have the value 'bind' or 'unbind'. Once bound, however, the BPRole acts as a reference to the Object to which it is bound, and as such may use the names of Features of the Object, and Associations in which it participates, just as with an instance of any other Class.

## 4.1 Introduction

The first part of this Chapter introduces the notation for drawing EDOC specifications according to the Process and Role Model and the Event Model. The second part of this Chapter introduces a number of patterns of usage, some of which have suggested notations that make them simpler to depict.

## 4.2 Activity and BPRole

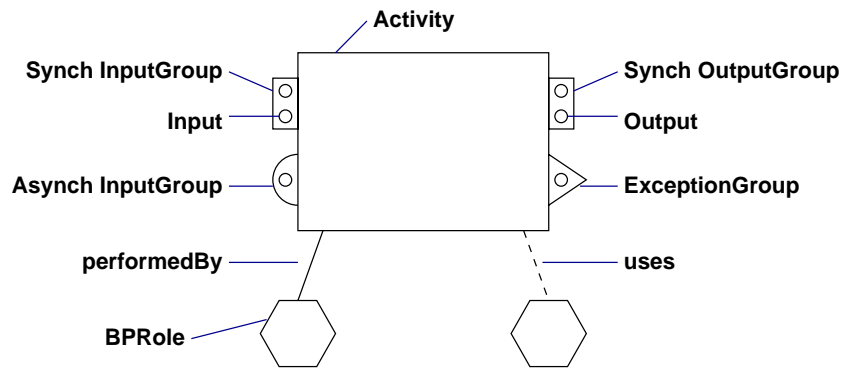


Figure 4-1 Activity with synchronous and asynchronous InputGroups, an OutputGroup and an ExceptionGroup.

As shown in Figure 4-1, an Activity is represented as a rectangle with protruding tabs. If the Activity has `definedBy` association to a CompoundTask then the rectangle has a drop-shadow as shown in Figure 4-2.

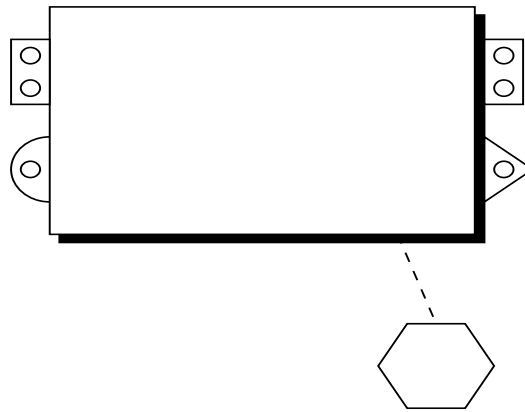


Figure 4-2 Activity defined by a Compound Task

InputGroups and OutputGroups associated with the Activity are depicted with special tabs attached to the left hand side of the task box for inputs and the right hand side for outputs. Specifically, rectangular tabs are used to indicate synchronous DataGroups, rounded tabs are used to indicate asynchronous DataGroups and triangular or beveled tabs are used to indicate ExceptionGroups.

DataElements (i.e., Inputs and Outputs) are represented as circles appearing in InputGroups and OutputGroups.

BPRoles are drawn as hexagons and are associated with Activities by either the `performedBy` association for performer roles, or the `uses` association for artifact roles. These associations are drawn as a solid line for the `performedBy` association, and a dashed line for the `uses` association. See Section 2.15 for more detail on the definition and usage of BPRoles. It should be noted that a single BPRole 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 `definedBy` association to a CompoundTask may not have a `performedBy` association to a BPRole.

Due to the derived inheritance from `UML::Foundation::Namespace`, (see Figure 2-1) all model elements can be uniquely named. Naming of model elements conforms to standard UML notation conventions.

### 4.3 *CompoundTask*

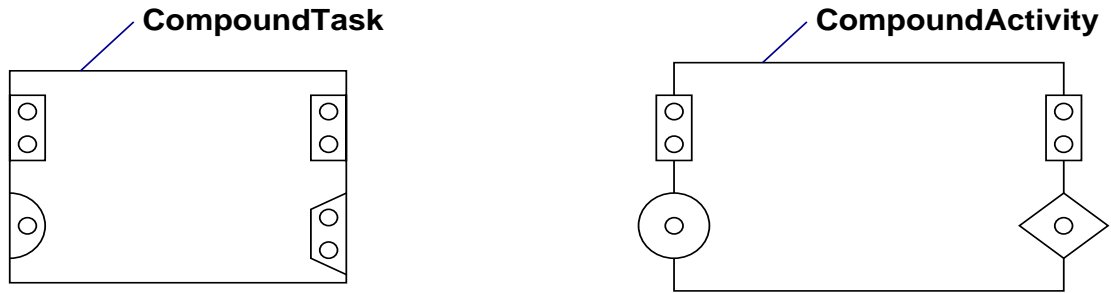


Figure 4-3 CompoundTask and ‘CompoundActivity’ notation.

As shown in Figure 4-3, a CompoundTask is represented as a rectangle with interior tabs. One may wish to depict an Activity and a CompoundTask’s internal details simultaneously in which case, the ‘CompoundActivity’ notation can be used.

As defined in Section 2.11, 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 BusinessProcess. Figure 4-4 shows how a CompoundTask containing a number of Activities and DataFlows is depicted.

Note that DataFlows are shown as solid lines with directional arrows linking source and sink DataElements. The connections of DataFlows are subject to the restrictions outlined in Section 2.13.

DataElements with no associated type (known as ControlPoints) are shown as solid dots.

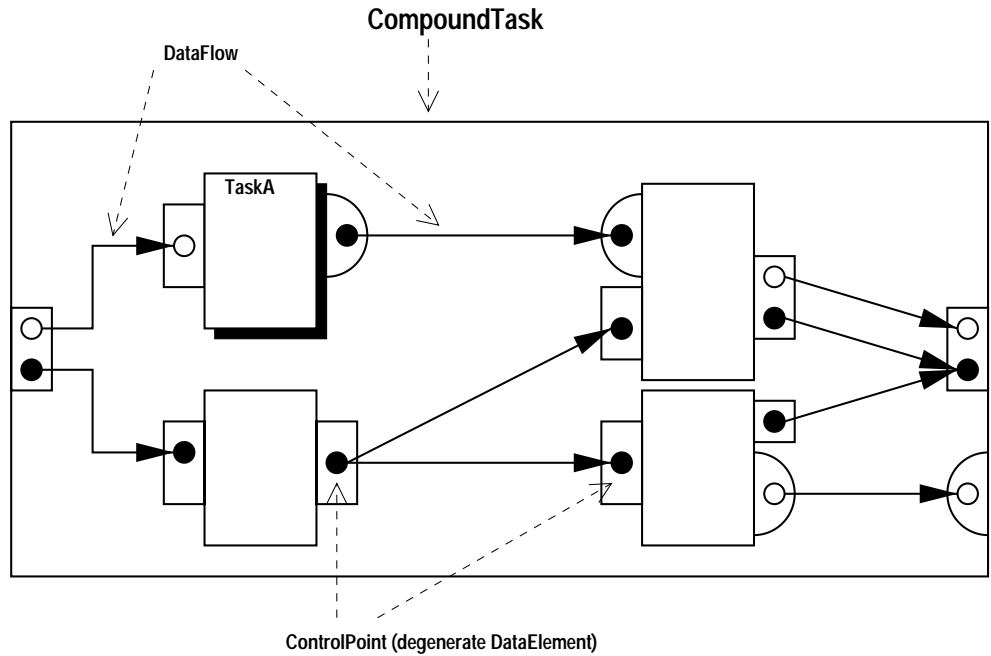


Figure 4-4 CompoundTask with Data Flows

In Figure 4-4 the Activity TaskA is an example of an Activity for which there is an associated CompoundTask but it has not been shown at this level of detail. The detail for TaskA would be shown in a related diagram.

#### 4.4 DataMap

As CompoundTasks are reusable, we only define the “inside” part of the DataGroups in a CompoundTask, leaving the “outside” part (and possible map between them) to the containing Activity which invokes the definition. Figure 4-5 shows how the existence of DataMaps are depicted using a thicker black line on the Activity side of a DataGroup tab.



Figure 4-5 DataMaps between an Activity and a CompoundTask.

If the mapping is the default one-to-one mapping of DataElements with the same name, then simple adjacency of DataGroups indicates this and the thicker black line can be omitted.

Figure 4-6 shows a UML-style representation of a DataMap. Typically this would not be shown in a diagram but would be accessible via a popup window in a tool. However, when it is useful to depict this level of detail this is the suggested notation.

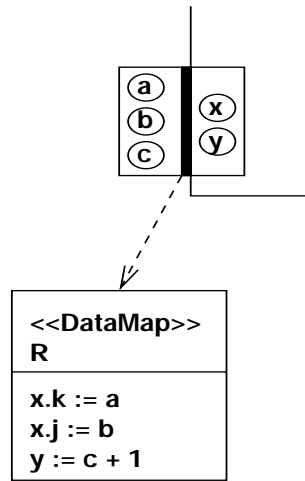


Figure 4-6 Detail of a DataMap which would not normally be shown directly in an EDOC Process diagram.

## 4.5 Events

Business Events are shown in the Process diagrams by using the notations shown in Figure 4-7. A zig-zag line is attached to the visual representation of the model element to be the source or sink of the EventSource or EventSink. The zig-zag line has a directional arrow head with outgoing arrows indicating EventSources, and incoming arrows indicating EventSinks.

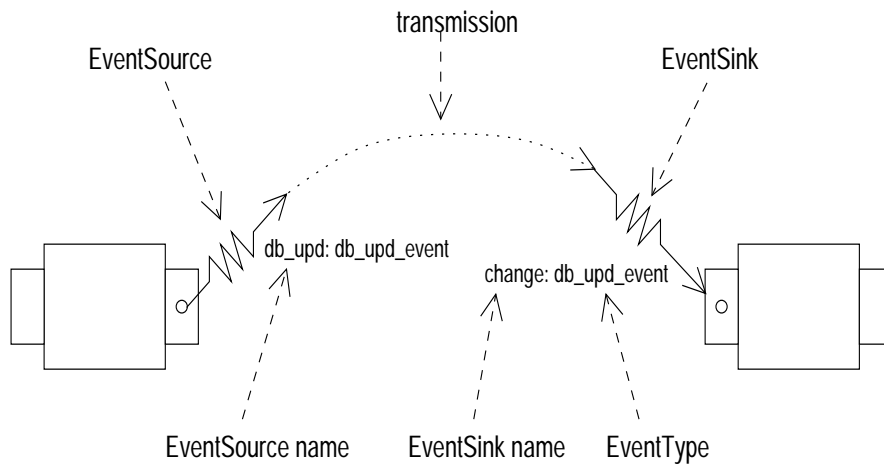


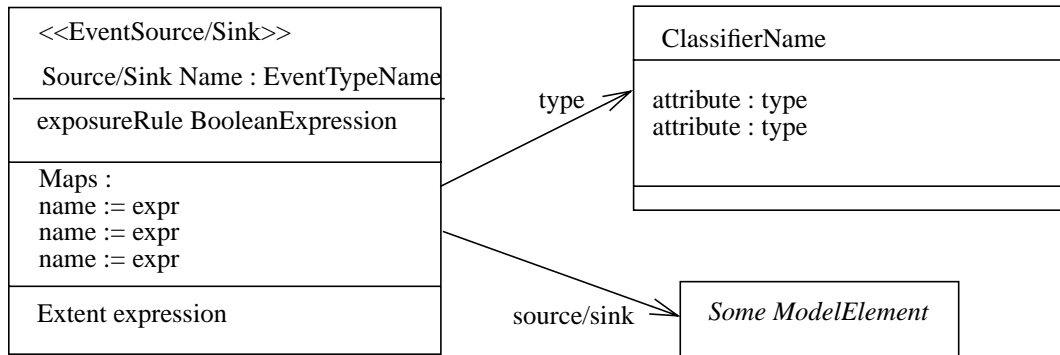
Figure 4-7 Business Event Notation.

For example, in Figure 4-7 the EventSink named *change* is attached to the synchronous InputGroup of the right-hand Activity while the EventSource named *db\_upd* is attached to the Output of the left-hand Activity.

## 4.6 Notation for Event Sources and EventSinks

We adopt the modified UML Class notation depicted in Figure 4-8 for describing the details of EventSources and EventSinks. This allows for the addition of details for describing the EventSource/Sink name, the Event type, the `exposureRule` BooleanExpression, the map Expressions, and the EventSource/Sink's extent.

### EventSource/Sink form:



### EventSink Example:

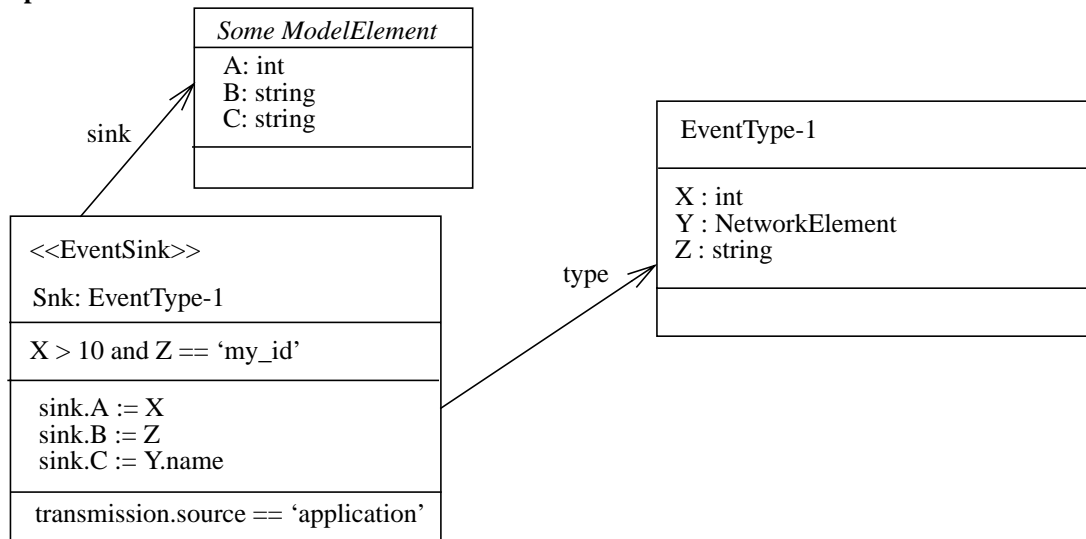


Figure 4-8 EventSource/Sink Detail Notation

## 4.7 Patterns Overview

The rest of this Chapter describes various patterns of common usage and associated special notation that may be useful when using the Business Process Model. We first describe the pattern in terms of its normal notation, possibly with templated 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 behaviour, especially in conjunction with concurrent threads. It is for this reason that we only allow Activities with synchronous DataSets to execute once. 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 while we occasionally use UML template notation for describing patterns, it is not sufficient in general since the patterns are usually parameterized by an Activity which will have some unknown number of InputGroups, Inputs, OutputGroups and Outputs which will need to be reflected in a pattern's CompoundTask and connected up with Flows accordingly.

## 4.8 Timeout

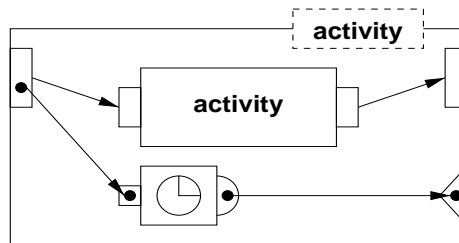


Figure 4-9 Templated activity with timeout.

Often we will want to have an Activity timeout after some period. The pattern shown in Figure 4-9 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 4-10 . This notation may also include a duration or absolute parameter which would be provided as input to the underlying timer activity.

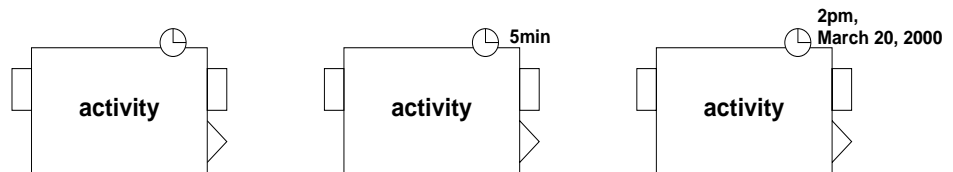


Figure 4-10 Timer pattern notation.

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 particular mappings to provide a suitable implementation.

### 4.9 Terminate

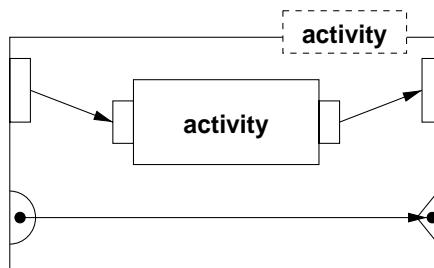


Figure 4-11 Templated activity supporting a terminate message.

We may wish to be able to terminate an Activity before it has completed of its own accord. The pattern shown in Figure 4-11 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 labelled asynchronous InputGroup and corresponding ExceptionGroup on any arbitrary Activity.

4.10 Guards

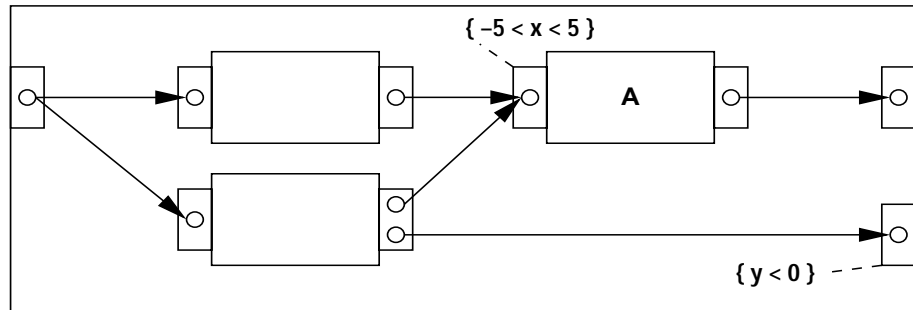


Figure 4-12 Guards on an InputGroup and an OutputGroup.

Sometimes it may be desirable to add a guard condition 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 4-12 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)..

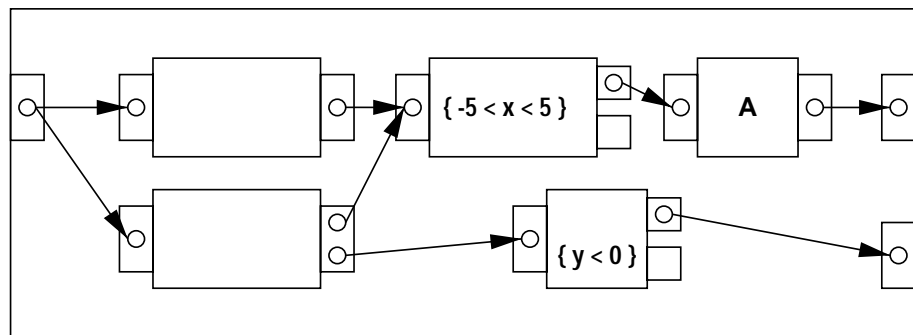


Figure 4-13 Equivalent model to Figure 4-12 using condition tasks.

Figure 4-13 shows an equivalent CompoundTask to that of Figure 4-12 but using explicit filter Activities that have 'success' and 'fail' OutputGroups.

This deals nicely with the case when none of the guards are satisfied and there are no more values available that might satisfy them. As can be seen from Figure 4-13, 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.

## 4.11 Simple Loop

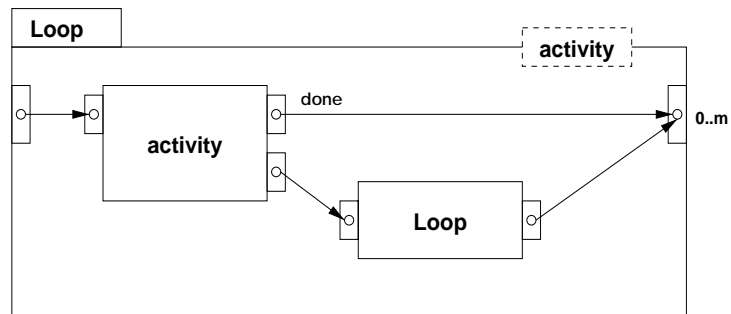


Figure 4-14 A simple loop using recursion.

The pattern shown in Figure 4-14 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 4.12 through Section 4.14 will be used.

A special-case shorthand notation for such a loop is shown in Figure 4-15. The looping flow indicates that simple recursion is taking place. Any OutputGroup containing an Output which is the source of a looping flow may only be the source of flows to a single InputGroup.



Figure 4-15 Special-case notation for a simple loop.

4.12 While and Repeat-Until Loops

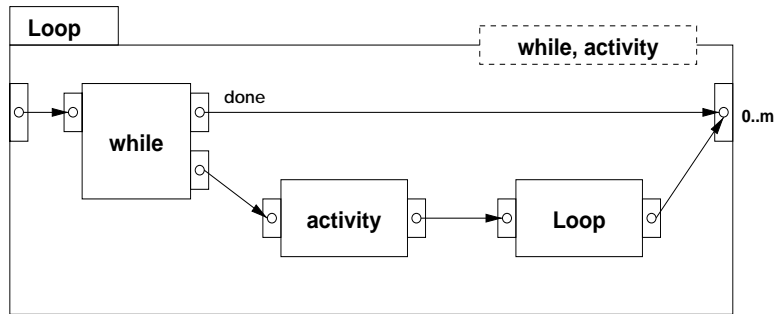


Figure 4-16 The while loop pattern.

In Figure 4-16 we see a more general ‘while’ loop pattern with separate exit test and loop body, and Figure 4-17 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.

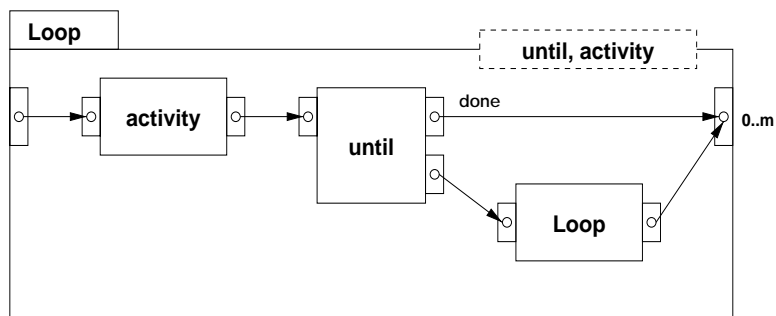


Figure 4-17 The repeat-until loop pattern.

As for the Simple Loop, these loops could be drawn as shown in Figure 4-18 and Figure 4-19 respectively.

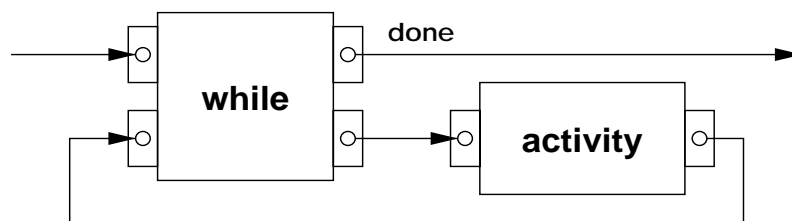


Figure 4-18 Special-case notation for a while loop.

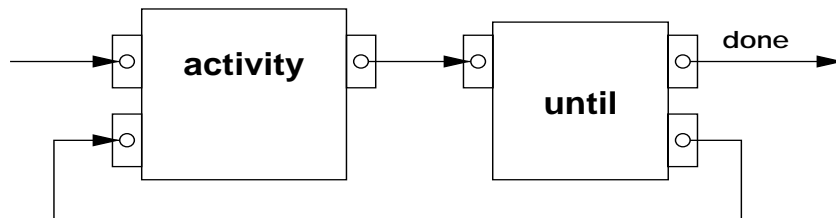


Figure 4-19 Special-case notation for a repeat-until loop.

## 4.13 For Loop

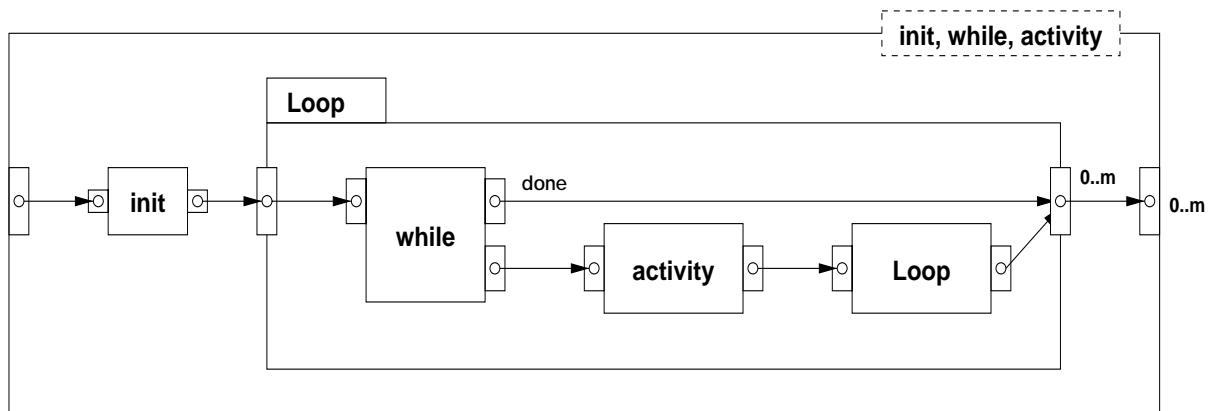


Figure 4-20 The for loop pattern.

The pattern in Figure 4-20 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.

## 4.14 MultiTask

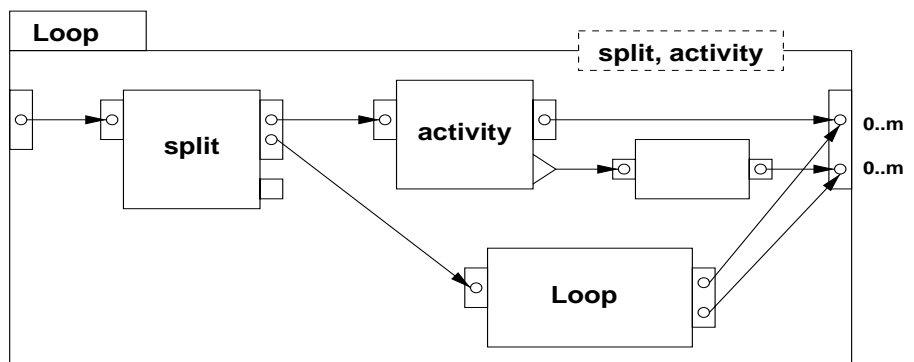


Figure 4-21 Pattern for a multi-task.

The pattern in Figure 4-21 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 process 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.



## *Example EDOC Specification*      *Appendix A*

---

### *A.1 An Example Specification of an Enterprise System*

The following pages contain a specification of a system for describing and supporting the processes for procuring goods or services for an organisation. An informal textual description of the system is given followed by specifications of the business processes, business entities, rules and events involved in this system.

This example has been developed in collaboration with an industry partner and represents the expression of the business processes used by the company for sourcing goods.

### *A.2 The Procurement System - An Informal Description*

The procurement system is concerned with the procurement of goods or services by an organisation.

The process for acquiring some resource (or service) can be started in either of two ways. In both cases, a request listing the resource requirements is received. In one case this is sufficient, however in the second case, the request is accompanied by additional information about the preferred freight options for delivery.

In both cases, the resource requirements are used as a basis for sourcing a number of potential suppliers of the goods. This list of potential suppliers (and for the second case, a corresponding list of freight sources) is then evaluated. The evaluation is a sophisticated process involving ranking and checking of potential suppliers. As a result of the evaluation, a supplier is awarded the contract to supply the required goods. Both the sourcing of potential suppliers and the evaluation process are the responsibility of the Purchasing Officer.

# Example EDOC Specification

After the Authorising Officer has awarded the contract to a particular supplier, the order is released to that supplier for processing. While the order is being processed, it is monitored to ensure that progress is made and the contract is fulfilled. Finally after the resources are received, the receipt of the goods is approved, and any claims for payment are fulfilled.

## A.3 The Business Process Model

The Procurement Business Process as shown in Figure A-1 provides for Resource Requirements to be satisfied from various sources for a given request.

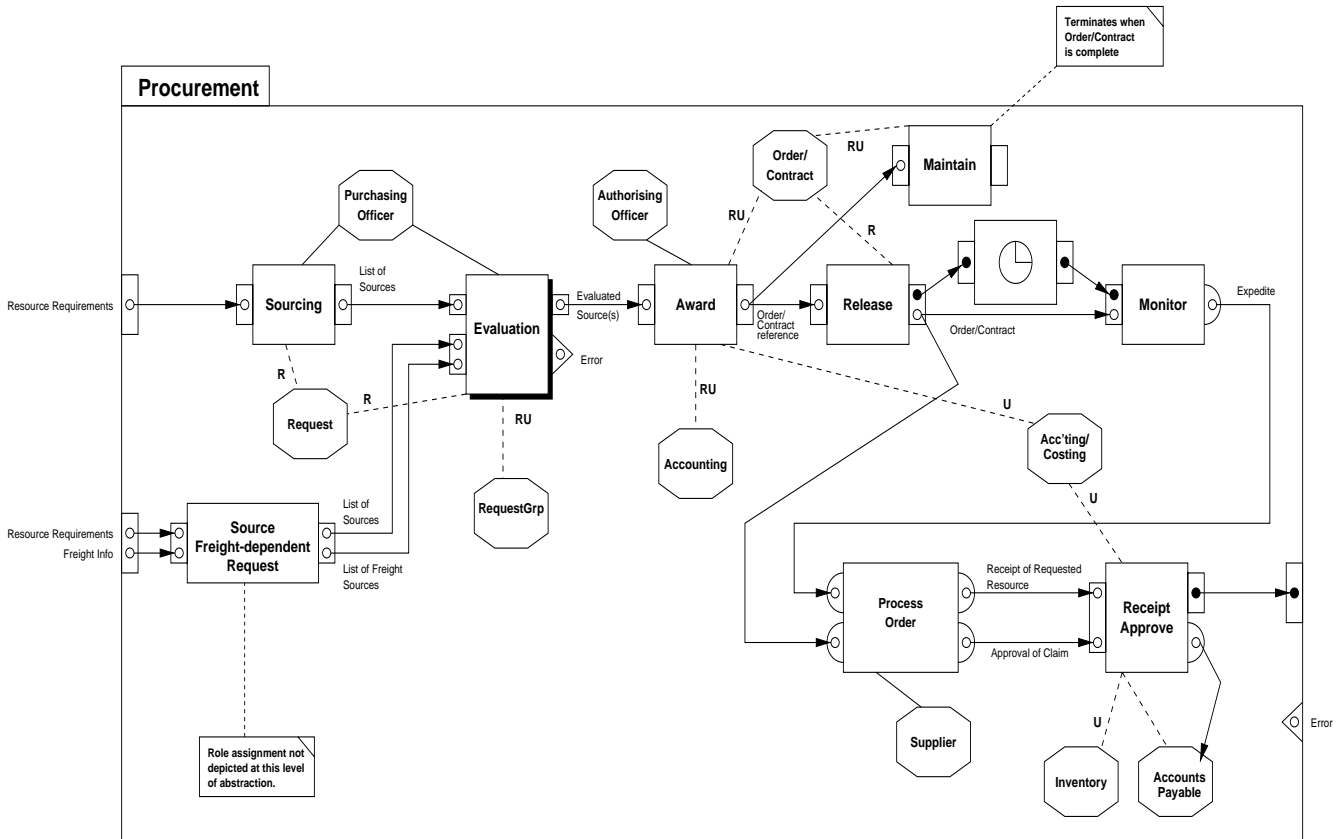


Figure A-1 Procurement Business Process

The Procurement Business Process can be initiated in one of two ways by the enabling of the two alternative input sets specifying

- Resource Requirements, or
- Resource Requirements plus Freight Requirements information if the request includes freight requirements.

The process completes successfully once sources for satisfying the Resource and Freight Requirements have been identified and evaluated, a contract has been awarded, released and processed, and finally the goods have been received and paid for.

Where no valid source can be found to satisfy the resource or freight requirements, the process will throw an appropriate user-exception indicating this and the process will terminate unsuccessfully.

The Procurement Business Process is Modeled as being comprised of a number of Activities and CompoundTask definitions. These are discussed in detail in the following sections.

### A.4 Detailed Task Description

Unless otherwise mentioned, all the following sections will refer to Figure A-1.

#### A.4.1 Sourcing and Sourcing Freight-Dependent Request Processes

Both the Sourcing and the Sourcing Freight-Dependent Request processes fulfill the task of determining a list of potential sources for satisfying the Resource Request. Both processes will reference sourcing policies applicable to the request as well as referencing the Request itself. The association to the Request BPRole is shown as a *uses* relation - that is, the request is referenced as an artifact role. This relation is annotated with an “R” to indicate that the access is a read-only operation.

The only distinction between the two tasks is that the Sourcing Freight-Dependent Request process has the additional work of considering the freight-requirements specified in the additional input to the task. Correspondingly it produces a list of sources for freight in addition to the list of sources for satisfying the resource request.

#### A.4.2 Evaluation

Having identified appropriate potential sources of supply, an evaluation is performed in accordance with the sourcing policy. This evaluation process completes successfully with output of the recommended preferred sources in a list ordered by the prioritised ranking of each source. The task can terminate with an exception when no valid sources can be found.

## Example EDOC Specification

The CompoundTask definition for the Evaluation Activity has been elided from the Procurement process model for clarity of expression. It presents more fine-grained detail than the rest of the Procurement processes and rather than show this extra detail in the Procurement process, it is removed to the separate diagram Figure A-2.

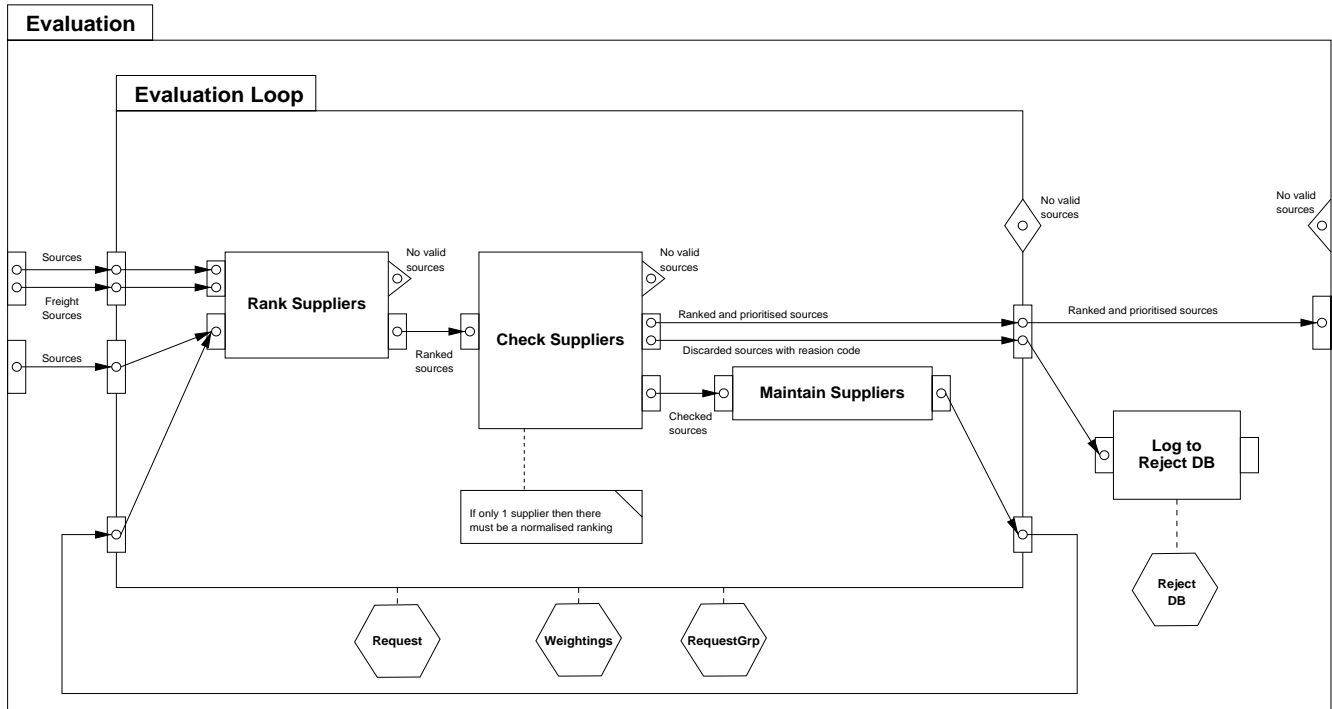


Figure A-2 Evaluation CompoundTaskDefinition

The Evaluation process is modeled as comprising an Evaluation Loop which iterates over a list of potential sources until either a prioritised list is produced, or the process is unable to find any valid sources and terminates with an exception.

The Evaluation Loop has three InputGroups. Two are for inputting the list of sources, and the list of sources accompanied by the list of freight sources. The third InputGroup has an Input that is a list of maintained and evaluated sources that will be subject to further evaluation.

The Evaluation Loop has two OutputGroups. The first has two Outputs - a list of ranked sources and a list of discarded sources. The list of discarded sources Output is itself Input to the Log to Reject DB process which records details regarding the rejection of sources. The second OutputGroup of the Evaluation Loop has a single Output that is a list of maintained or altered and evaluated sources that will be subject to further evaluation. This Output is connected by a DataFlow back to one of the InputGroups allowing for re-iteration over the list of sources.

The Evaluation Loop makes use of a number of Artifact BPRoles. It uses the Request, Weightings of the sources, and possibly the Request Group that is related to a specific request to assist in the evaluation. The Evaluation Loop terminates when all of the potential Sources have either been ranked and prioritised, or added to the list of discarded sources.

### *A.4.3 Award*

The Award process takes as Input the list of Evaluated Sources. From this list, the selected supplier is assigned to the Request and an order, contract or contract release is created as an Artifact BPRole. The Activity produces as Output a reference to the Artifact BPRole representing either the order, the contract, or the contract release.

The Award Activity is performed by the Authorising Officer BPRole.

Commitment details about the order, contract or contract release are passed to the Accounting artifact BPRole.

Both the Maintain and the Release Activities may start concurrently after the Award Activity has enabled it's output as they are both connected by DataFlows from the Output of this Activity.

### *A.4.4 Maintain*

The Maintain Activity supports the maintenance of the Orders, Contracts or Contract Releases. It takes as input an identifier for an Order, Contract or Contract Release and uses this reference to read and possibly modify the actual data. The Maintain process uses the identified Order, Contract or Contract Release as an artifact BPRole. Basically this process exists in recognition that Order, Contract or Contract Release are not completely static or stable and will need modification due to unforeseen circumstances.

This process has no Outputs.

### *A.4.5 Release*

The Order/Contract or Contract Release is forwarded to the selected supplier as part of the Release Activity. The Activity takes an identifier for an Order, Contract or Contract Release as Input and passes this identifier on as an Output.

The termination of the Release Activity enables a Timer Task to start when it receives a signal via a Control Flow from the OutputGroup of the Release Activity. The Timer Task is used to enforce a delay on the enabling of the Monitor Activity.

### *A.4.6 Monitor*

The Monitor Activity provides mechanisms to monitor supplier performance for timely delivery of the goods or services. It also monitors compliance with the terms of the Order, Contract or Contract Release.

The process has a single Asynchronous Output that will be some notification to the supplier to expedite delivery.

### *A.4.7 Process Order*

The Supplier BPRole performs the Process Order Activity. The Process Order Activity represents the actual supply of the goods or services to satisfy the order. The Supplier can be considered to be always active and waits for asynchronous notification of an order as a trigger to start processing that order. The Supplier is also able to asynchronously receive notifications from the Monitor Activity requesting that the Supplier expedite the process of satisfying the order.

Asynchronously, the Supplier will supply goods to satisfy the order and will also generate notification of invoices that require payment for the delivery of the goods.

### *A.4.8 Receipt and Approve*

The Receipt and Approve Activity handles the receiving goods, updating the inventory to reflect this, and the payment of invoices for the goods.

This process has an InputGroup comprising of a details relating to the receipt of the requested resource, and a request for approval of a claim for payments from the supplier.

### *B.1 EAI Mapping for the Business Process Model*

This section shows how the BPRoles, Activities and DataFlows of an EDOC Process Model can be realized using the Common Model Elements in the UML Profile for Event-based Enterprise Application Integration. (Based on the Collaboration Profile in the initial submission by Concept Five Technologies, International Business Machines, Oracle, Rational Software and Unisys: ad/00-08-05 and ad/00-08-12.)

The basic approach is that each Legacy Application is represented in the Process Model as a BPRole which performs some of the Activities in the Process. Conversely, an Activity represents some action performed within the Legacy Application upon receipt of DataFlows by one of its InputGroups.

DataFlows are mapped to Messages in the EAI context. The DataElement from which a DataFlow emanates is mapped to an EAI Source, while the DataElement at which a DataFlow terminates is mapped to an EAI Target.

Aside from the basic mapping of DataFlows to Messages and the DataElements which they connect to Sources and Targets, there are two possible mappings of the rest of the Business Process Model. Each model element's specific mapping that is impacted by this choice will include a section on common mappings, as well as an (A) and (B) alternative that express the following:

#### ***(A) Mapping without coordination semantics***

The generated EAI model will not contain the coordination semantics for Messages that the EDOC Process model embodies. The EAI model will simply contain the message type and interconnection architecture which will facilitate the DataFlows shown in the Process Model. The behaviour of the Sources and Targets upon receipt of the messages representing DataFlows will need to conform to the Process Model.

This approach is useful when the InputGroups and OutputGroups defined on the Activities in the Model are simple, and correspond directly to the way in which values are emitted and consumed by the legacy applications being integrated. In general this

means that Activities will probably have a single InputGroup with a single Input that is an aggregation of the values needed to perform the Activity. The Activities would also have simple OutputGroups with single Outputs.

Alternatively the mapping from the generated EAI profile to the implementation technology will provide an implementation of connectors to the legacy applications that embody the semantics of an EDOC Process's InputGroups and OutputGroups.

### ***(B) Mapping with coordination semantics***

The generated EAI model will contain Operators that embody the semantics of the mapped InputGroups and OutputGroups contained in the EDOC Process Model. The structure of the generated model will be far more complex, and contain many more model elements. But with the correct implementation of the specified Operators, the mappings to implementation technologies will require no special knowledge of the coordination semantics of the EDOC Process Model.

### ***B.1.1 BPRole***

Business Process Roles (BPRoles) in the Process Model are mapped to ClassifierRoles in the EAI Collaboration Model. Each BPRole that is associated with an Activity is denoted by a stick figure icon, and named after the `find` attribute of the BPRole (of type `UML::ObjectSetExpression` - a string) which is used to identify the legacy application. The type of the ClassifierRole's base Classifier is the same as the type of the Classifier denoted by the BPRole's `type` association.

Each ClassifierRole generated will be connected via undefined Interactions to the Sources and Targets generated for each of the Inputs and Outputs of the Activities performed by the BPRole from which it was mapped. (See Figure B-1 on page 75, and "DataGroup" on page 77.)

---

Note – EDOC BPRoles have `find` and `factory` attributes containing expressions that are used to locate or create an appropriate object to perform the Activity to which is it associated. In the EAI context Legacy Applications are assumed to have a unique name. This means that several BPRoles with different names may map to the same ClassifierRole if they contain the same `find` expression. The EDOC Process Modeler may choose to use a single BPRole to represent a single Legacy Application, but may also use several BPRoles with different names, but containing the same `find` expression.

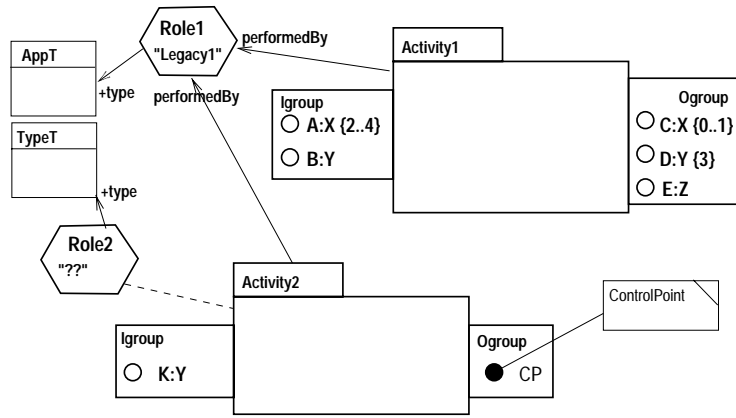
---

### ***BPRoles performing Activities***

Each ClassifierRole that represents a performer of an Activity (performer role) will have an Interaction with the Sources and Targets generated by the mapping of the Inputs and Outputs of that Activity. These Interactions will be refined by the EAI designer. See "Role2" in the example in Figure B-1 on page 75.

## BPRoles used by Activities

The ClassifierRoles representing BPRoles that are used by an Activity (artifact role) will be connected to the ClassifierRole representing the performer of the Activity by an undefined Interaction. This Interaction will be refined by the EAI designer. See “Role1” in the example in Figure B-1 on page 75.



maps to (partial)

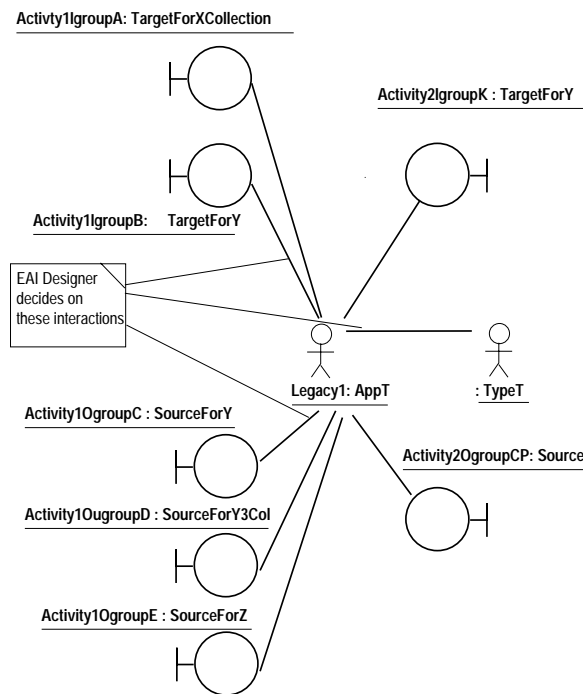


Figure B-1 EAI Mapping for Role and DataElement

### B.1.2 Activity

An Activity is the encapsulation of some action resulting from an InputGroup owned by the Activity being enabled. The performer BPRole associated with the Activity by a `performedBy` association will identify the application that will receive the values of that enabled InputGroup. This application will also produce values that it will transmit to the Outputs of one of its synchronous OutputGroups (or any of its asynchronous OutputGroups).

The Activity is mapped by generating a ClassifierRole for the BPRole that the Activity is `performedBy`. The DataElements in the InputGroups and OutputGroups owned by the Activity are mapped to Sources and Targets that are connected to the ClassifierRole via unspecified Interactions. The mapping for the DataGroups owned by the Activity are specified in the Sections below.

### B.1.3 DataElement

This abstract type is the supertype of Inputs and Outputs. The mappings for these two subtypes generate different EAI Profile model elements. In addition they both generate some model elements in common for mapping (B).

**(A)**

See the concrete subtypes. No additional elements are generated.

**(B)**

See the concrete subtypes. In addition each DataElement in a synchronous DataGroup is mapped to a Transformer within a CompoundOperator that represents the DataGroup which owns the DataElement. See “DataGroup” on page 77. See also Figure B-2 on page 79.

### B.1.4 Input

An Input to an Activity is mapped to a `TargetFor<typeof(Input)>`, and an Interaction between this Target and the ClassifierRole representing the performer of the Activity. This Interaction will be unspecified and must be refined by the EAI designer. See Figure B-1 on page 75 for an example.

An Input to a CompoundTask is mapped to a `SourceFor<typeof(Input)>`, and an Interaction between this Source and the ClassifierRole representing the CompoundTask. This Interaction will be unspecified and must be refined by the EAI designer. See Figure B-4 on page 83 for an example.

### B.1.5 Output

An Output of an Activity is mapped to a `SourceFor<typeof(Output)>`, and an Interaction between this Source and the ClassifierRole representing the performer of the Activity. This Interaction will be unspecified and must be refined by the EAI designer. See Figure B-1 on page 75 for an example.

An Output of a CompoundTask is mapped to a TargetFor<typeof(Output)>, and an Interaction between this Target and the ClassifierRole representing the CompoundTask. This Interaction will be unspecified and must be refined by the EAI designer. See Figure B-4 on page 83 for an example.

### B.1.6 Control Point

A Control Point is simply an untyped Input or Output that is used to sequence the beginning of one Activity after the end of another without transmitting any specific data. It has a special EDOC graphical representation, but no special metaclass in the EDOC model. ControlPoints are mapped to generic Sources and Targets in the same way as Inputs and Outputs are mapped to typed Sources and Targets. SeeSection B.1.4 andSection B.1.5 above for details, and Figure B-1 on page 75 for an example.

### B.1.7 DataGroup

Each DataGroup has a boolean attribute `sync` that indicates, when true, that the Activity to which the DataGroup belongs is synchronized to begin after (in the case of an InputGroup) or end before (in the case of an OutputGroup) the DataGroup is enabled. When false, it indicates that the Activity is already (in the case of an InputGroup) or still (in the case of an OutputGroup) running when values are passed to (from) the enabled DataGroup.

#### *Synchronous*

A Synchronous DataGroup either never sends any results from its DataElements, or makes all the results available on all its DataElements logically simultaneously. This notion is analogous to the return value and output parameters of a method call becoming available to the caller upon completion of the method's action. However, in a message passing, asynchronous implementation design framework, such as the EAI Profile, the individual copies of the results that flow from the DataElements to one or more other DataElements will not be transmitted at exactly the same time.

In mapping A, we assume that the legacy system that is identified by the Role performing the Activity which owns the DataGroup will actually have all the results for a single synchronous DataGroup available before beginning to transmit them. (And that only one synchronous DataGroup will ever transmit a set of results.)

In mapping B we introduce a Compound Operator which correlates DataElements for a DataGroup.

(A)

The ClassifierRole created for the performer of the Activity owning a DataGroup will have a direct Interaction with the Targets and Sources generated by the mapping of the Inputs or Outputs owned by an OutputGroup. See "Input" on page 76. & See "Output" on page 76.

(B)

A DataGroup is mapped to a CompoundOperator with the same name, and of the following form:

Each DataElement of the DataGroup is mapped to a Transformer. The output Terminal of these Transformers is of a “Union” type derived from the names and Types of the DataElements contained by the DataGroup. The “union” type is a new class named by concatenating the names of the types of the DataElements, separated by “or”. The new class then has an XOR constraint over a set of aggregations, one named for each of the DataElements in the Group. See Figure B-2 on page 79 for an example - “XorYorZ” which is the “union” type for 3 DataElements of types X, Y and Z.

---

Note – This naming scheme may lead to conflicts in some mappings. A better naming or scoping scheme is probably required.

---

A Router is introduced by the mapping to implement the semantics of the DataGroup’s synchronization function. Its Input Terminal is of the same type as the Output Terminals of the Transformers mapped from the DataElements. The Router will have an Output Terminal corresponding to each Transformer, with the same names and types as the Transformer’s Input Terminal. See Figure B-2 on page 79 for an example.

The Output Terminal of each Transformer is connected by an Interaction to the Router’s Input Terminal. The Compound Operator exposes all the Input Terminals of the Transformers, and all the Output Terminals of the Router. This gives it identical Input and Output terminals, with the exception of those terminals representing DataElements with multiplicities whose upper bound is greater than one. In this case the Input Terminal to the Transformer will accept single values of the type of the DataElement, while the corresponding Output Terminal of the Router will emit collections of this type.

---

Note – This mapping could be greatly simplified if Operators were allowed to have more than one Input Terminal; i.e. the “union” types would not be needed.

---

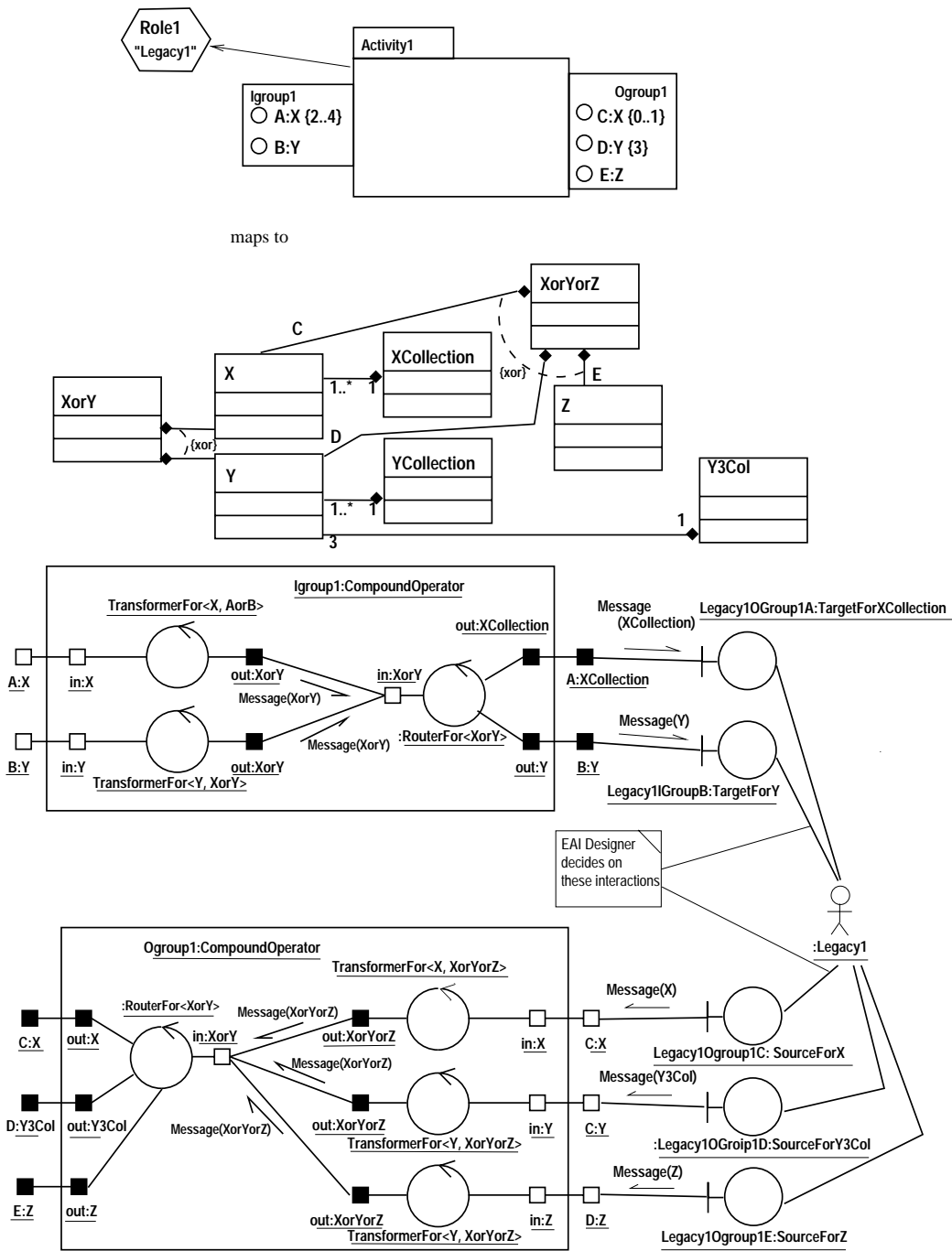


Figure B-2 EAI Mapping (B) for DataGroup

### *Asynchronous*

An asynchronous DataGroup represents the ability for an Activity to transmit results during its execution. However, the individual DataElements that make up the DataGroup are correlated in the same way as synchronous DataGroups. Therefore the mapping is identical, except that there is an assumption that the Activity can accept values through these asynch DataGroups only during the lifetime of the Activity, which is bounded by the enabling of any synchronous InputGroup and the enabling of any synchronous OutputGroup.

### *B.1.8 ExceptionGroup*

ExceptionGroups are the same as OutputGroups in behaviour unless they do not have DataFlows from their Outputs. In this case the semantics is that when an ExceptionGroup is enabled, the values are transmitted to the Standard ExceptionGroup of the containing CompoundTask, which terminates all its Activities and propagates the Exception message outwards.

This means that the generated mapping for EAI depends on the existence of DataFlow(s) whose sources are the Outputs of the ExceptionGroup.

---

Note – The mapping should use the EAI “ExceptionTarget” model element. Work to be done.

---

(A)

(B)

### *B.1.9 DataFlow*

DataFlows in the EDOC Process Model are mapped to Interactions in the EAI Collaboration Profile. For each DataFlow in the Process Model an Interaction is generated that occurs between the ClassifierRoles in the EAI Collaboration that are generated to represent the DataElements between which the DataFlow communicates.

---

Note – Not all the Interactions generated in this mapping are complete... some Interactions between Operators are not spelled out.

---

(A)

In this mapping the Activities whose DataElements are joined by DataFlows are represented in the EAI Collaboration Profile by ClassifierRoles whose implementation is aware of the semantics of the EDOC Processes. These Activities’ OutputGroups are

mapped to SourcesFor the types of their Outputs. The implementation of these Sources will be aware that they need to duplicate values that they receive for each Interaction that represents a DataFlow.

The DataFlows are mapped to Interactions labelled Message(<type>), where *type* is the mapped type of the DataElements (and hence the parameter to the Templated SourceFor and TargetFor Classifiers represented by ClassifierRoles in the Collaboration) which the DataFlow connects. The direction of the asynchronous arrow is the same as the direction of the flow.

**(B)**

In this mapping the synchronous DataGroups containing the DataElements between which DataFlows occur are mapped to CompoundOperators in the EAI Profile, while the mapping for asynchronous DataGroups are simply mapped to Sources and Targets. The EDOC Process Model semantics for multiple DataFlows emanating from a single DataElement are that whatever value(s) is (are) available when the DataGroup becomes enabled are copied and sent to all DataElements connected via a DataFlow.

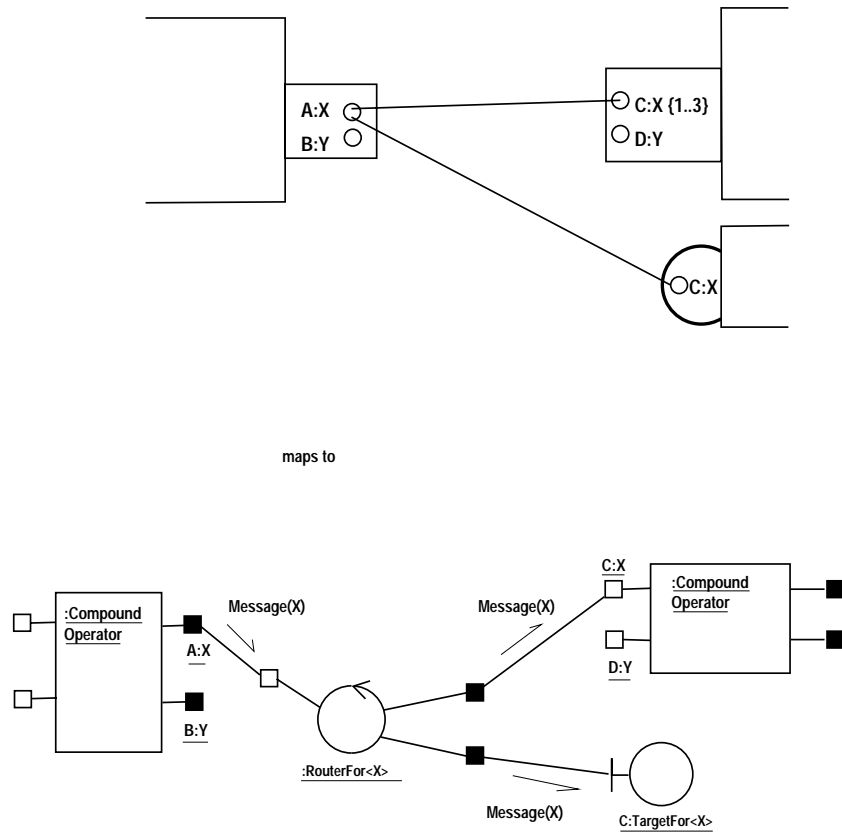


Figure B-3 EAI Mapping for DataFlow

For a given DataElement the semantics of multiple DataFlows emanating from it is implemented in the EAI Profile by the introduction of a Router and Interactions. The Router has an Input Terminal of the same type as the corresponding Output Terminal

of the CompoundOperator generated from a synchronous DataGroup (or Source or Target generated in the case of asynchronous DataGroups), and one Output Terminal for each of the DataFlows emanating from that DataElement. It has an Interaction labelled Message(<type>) between the Output Terminal (or Source or Target) representing the DataElement from which the DataFlow emanates, and the Input Terminal of the Router. The Router's Output Terminals are connected by identical Interactions to each of the Input Terminals (or Sources or Targets in the asynch case) representing the DataElements to which the DataFlows proceed. The type of the Messages sent over these Interactions is the same as the type of the DataElements that the DataFlows connect.

The behaviour of the generated Router is that for each Message it receives on its Input Terminal it creates a copy for each Output Terminal that it has and sends a copy to that Terminal, from whence it is transmitted via the Interaction to its destination.

### B.1.10 CompoundTask

A CompoundTask represents the initialization and termination of a set of coordinated Activities. As such, it will be mapped to a set of Sources and Targets and the Interactions that represent the DataFlows to and from its DataGroups (see (A) and (B) below). The mapping will generate a ClassifierRole that represents the initiator of an instance of the Process defined by the CompoundTask. For convenience this ClassifierRole will have the same name as the CompoundTask.

The mapping of the Inputs and Outputs belonging to the InputGroups and OutputGroups of the CompoundTask is converse to that of Activities. An Input to a CompoundTask is mapped to a Source which interacts with the ClassifierRole representing the Initiator of the CompoundTask, and an Output is Target for the results of the Activities contained in the CompoundTask to propagate their results back to the Initiator. See section B.1.3 for details.

Nested CompoundTasks that are referenced by Activities within an enclosing CompoundTask are mapped by generating a ClassifierRole in the same manner as those representing Activities to be enacted by a Role, but the represented Classifier of that ClassifierRole will be a subsystem that exposes interfaces generated as defined in the Model Interaction chapter.

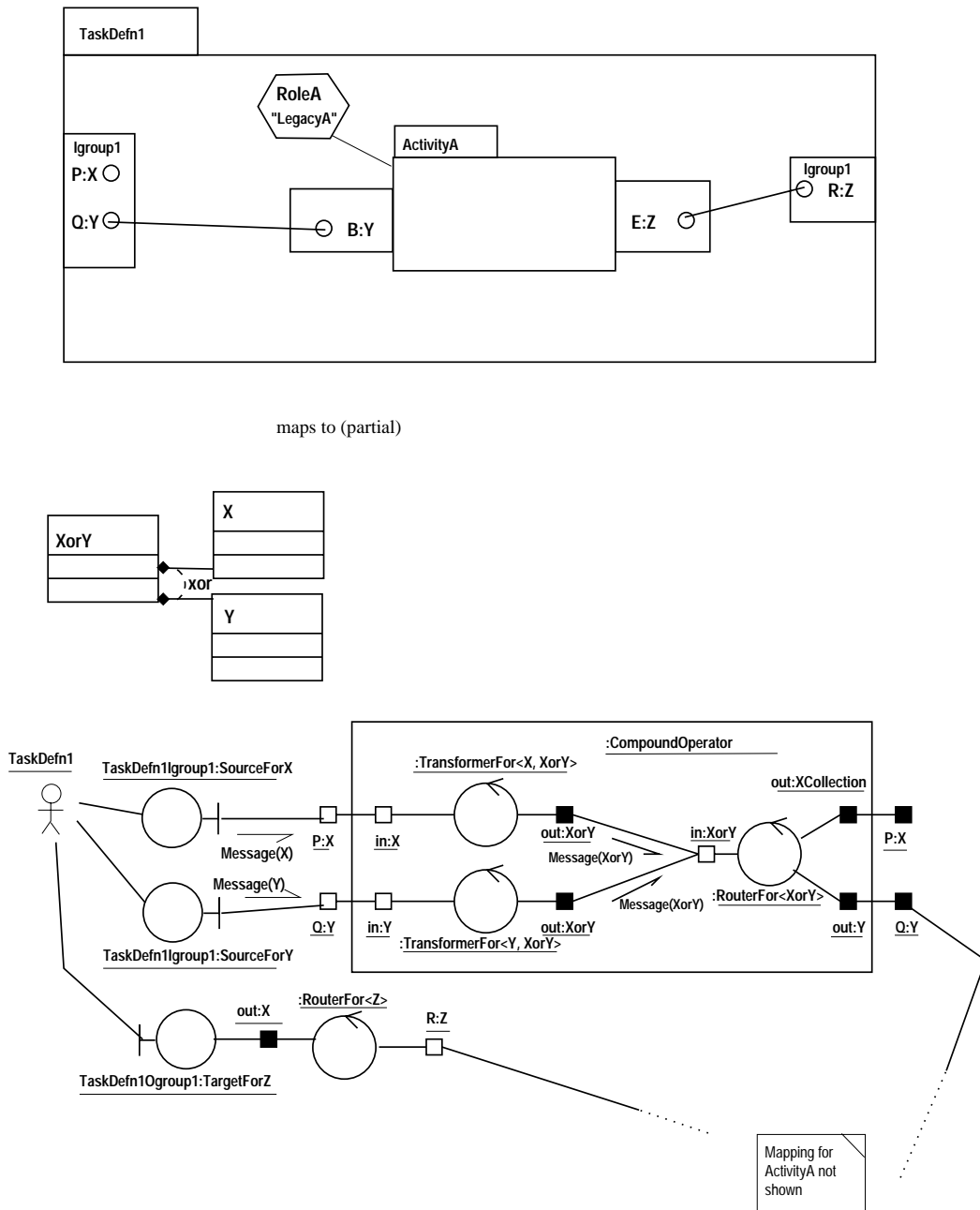


Figure B-4 EAI Mapping (B) for CompoundTask

## B.2 Process Model to Entity Model Mapping

There are two approaches to making an invocation of an Entity from within an Activity of a Business Process.

1. The bottom up approach. In this approach we assume that the Entities to be used to perform Activities in the Process Model are already designed. We expose all of the features of an Entity as Input and OutputGroups directly. The features that are actually required within the Business Process Model are then selected by placing the Activity in a Compound Task whose Input and OutputGroups represent only the features of the Entity to be exposed, and using a Data Map to convey the values of the InputGroups of the Compound Task to the corresponding InputGroups on the Activity, as well as mapping the appropriate OutputGroups of the Activity to those available on the Compound Task. The DataGroups of the Activity that are not mapped by the Data Map will be inaccessible to the user of the Compound Task. See Section B.2.7 “Generation of an Activity from Features of an Entity”.

The enabling of an InputGroup of the Compound Task would result in these values being transferred via the Data Map to the Activity, which would then automatically result in an invocation of the appropriate feature of the wrapped entity. If that feature is a method then the Activity would complete when the method returned, and the method result and out parameters would be conveyed directly to the matching OutputGroup of the Activity. These values would then be conveyed via a Data Map to the containing Compound Task. Asynchronous DataGroup satisfaction would similarly convey the data values to the appropriate asynchronous feature of the wrapped Object (or via a link to the Object representing an Event Sink).

2. The top down approach. In this approach we assume that a Process Model has already been designed, and the interactions with the Entity Model are chosen later. We use a Collaboration containing two “portal” ClassifierRoles one of which represents the InputGroups of the Activity and the other represents the OutputGroups of the Activity. This approach allows the designer to create additional ClassifierRoles within this Collaboration to represent arbitrary entities in the Entity model. The designer would then specify appropriate Interactions and Messages to cause the invocation of appropriate Entity behaviours resulting from invocation of the Input portal’s features which represent satisfaction of InputGroups. The designer would also specify Messages that caused invocation of Operations of the Output portal that represent the satisfaction of OutputGroups of the Activity. See Section B.2.8 “Generation of Skeleton Collaboration from an Activity”.

### B.2.1 Generic Mapping of DataGroup to Operation

A DataGroup is a collection of data values that represent the initiation or result of an action on a Task. The DataElements of a DataGroup are equivalent to parameters to this action. As such, each DataGroup is mapped to a UML Operation with the same name, and each Data Element that it contains is mapped to a UML Parameter of this Operation.

The type of each Parameter is determined by a combination of the type and multiplicity of the DataElement it represents. A DataElement with a multiplicity of {1} or {0,1} is represented by a parameter with the same type as the DataElement. DataElements with other multiplicities are represented by Parameters whose type is an Aggregation of the type of the Data Element, where the multiplicity of the Aggregation is the same as that of the Data Element. The `kind` attribute of the Parameters representing the Data Elements of the DataGroup is always “in”.

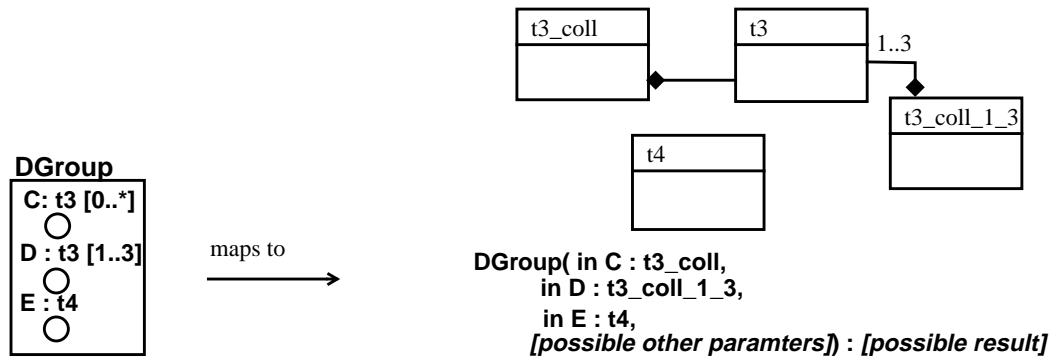


Figure B-5 Example mapping for a DataGroup

Additional parameters and a result are added to the generated Operation for the mapping of the concrete subtype InputGroup when it is owned by a CompoundTask (see section B.2.2 below for details).

## B.2.2 Mapping Synchronous InputGroups

The basic concept of this section is that the synchronous InputGroups of a Compound Task defining a Business Process are represented as Operations on a Factory Class that creates process instances. These instances are of a run time Business Process Class, also generated by this mapping. The instance created will inherit from a management Class, and it will be the entity to which Event Sinks representing asynchronous InputGroups will be attached.

Synchronous InputGroups of a CompoundTask that defines a Business Process, or of an Activity performed by a BPRole, are mapped as Operations on a Class. In the case of a BusinessProcess this Class represents the factory object for the Compound Task. In the case of an Activity the Class represents the “portal” (or viewpoint interaction) into the entity model, which allows a modeler to use a ClassifierRole, with this class as its base, in a Collaboration with other ClassifierRoles representing the BPRoles which perform or are used by the Activity. This is the hook needed to allow the modeler to use the values that arrive at an Activity’s InputGroups as parameters to interactions with the performer BPRole (and possibly some artifact BPRoles).

## Mapping InputGroups of a CompoundTask that defines a Business Process

The Class generated by this mapping is stereotyped <<BPFactory>>. The Operations follow the DataGroup mapping above, with the following additional parameters present:

- A Manager parameter whose kind is “return”, and its type is Classifier. This return value is a placeholder for a mapping to a particular technology to return a management interface for the Business Process. (This is shown as MyBP below, and the manager class is explained in section B.2.6).
- A Callback parameter whose kind is “in” and whose type is the “Output” Class mapped from the OutputGroups above. The caller of the operations corresponding to the InputGroups must implement a callback object conforming to this type so that the Business Process may return results corresponding to the synchronous OutputGroups and Exceptions of the Compound Task that defines it.

---

Note – Mappings to some technologies, for some short-lived Business Processes may optimise the return of results via an invocation of one of the operations corresponding to an InputGroup, by defining a Classifier that contains features representing the Outputs of the Business Process, and return this via the Manager “return” parameter. These implementations will accept a null Callback parameter.

---

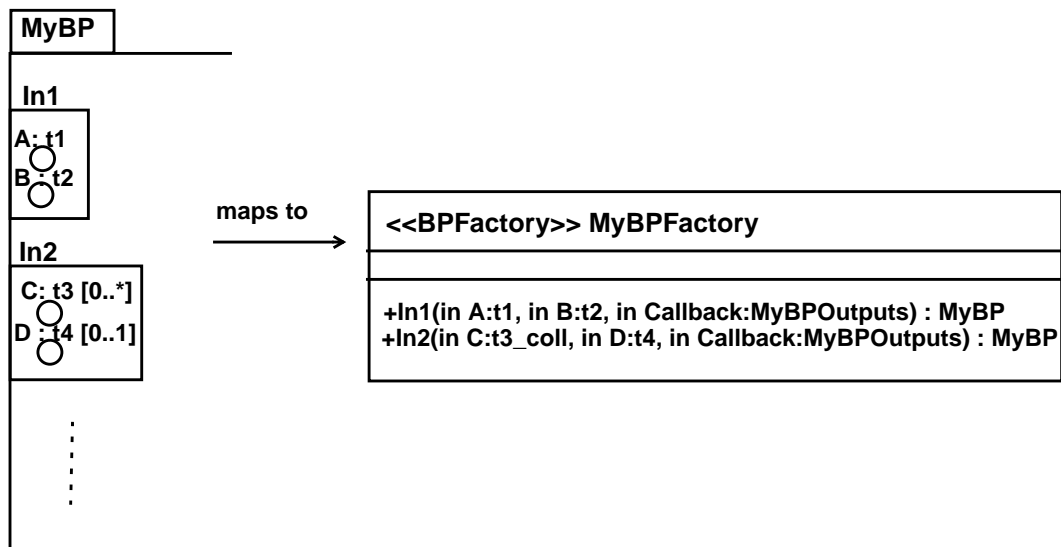


Figure B-6 Example mapping for the InputGroups of a CompoundTask to a Business Process Factory (see Figure B-10 on page 89 for the mapping of MyBP instances).

See Section B.2.6 “Mapping a Business Process as an Entity” for details of the MyBP manager Class.

## Mapping InputGroups of an Activity that is performed by a BPRole

From each Activity a Class is generated with the same name as the Activity, appended by “Inputs”. It is stereotyped <<ActivityInputs>>, and has Operations mapped from its synchronous InputGroups as defined by the generic DataGroup mapping in section B.2.1.

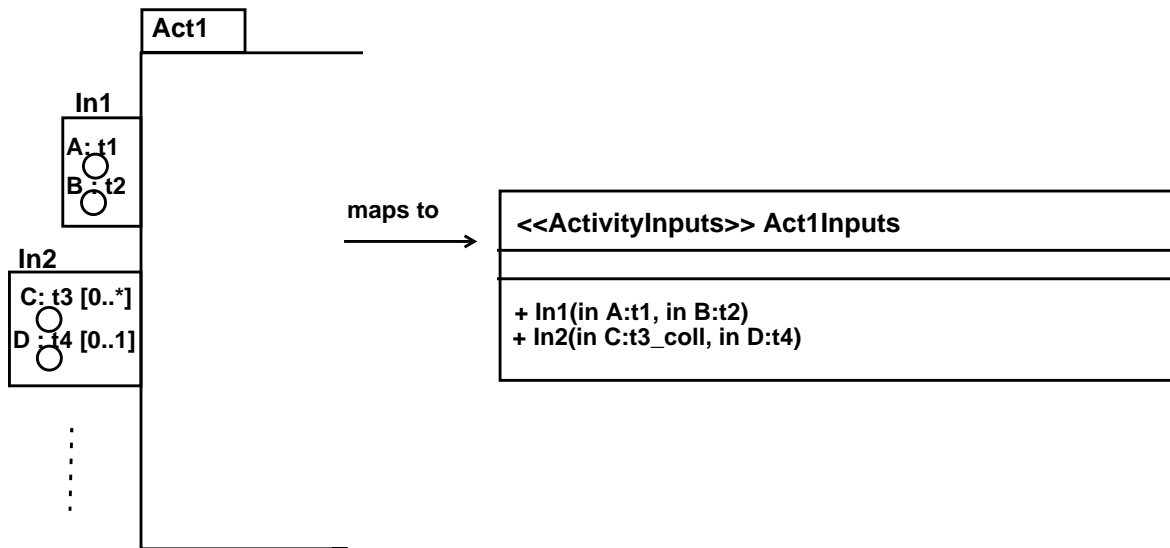


Figure B-7 Example mapping for the InputGroups of an Activity

## B.2.3 Mapping Synchronous OutputGroups

The synchronous OutputGroups are represented as Operations on a Callback Class which is implemented as an Object supplied by the Business Process user/creator. The Exceptions defined on the Compound Task are also represented as Operations on this Callback Class. The asynchronous OutputGroups become Event Sinks on the Callback Class.

The same Class generation mapping is used for the OutputGroups of a CompoundTask that defines a Business Process as for the OutputGroups of an Activity that is performed by a BPRole. Only the stereotype of the Class is different. All the OutputGroups of a Task are mapped as a Class with the same name as the Task, appended by “Outputs”. Each OutputGroup, including ExceptionGroups, is represented as an Operation mapped as described in the DataGroup mapping in section B.2.1.

## Mapping OutputGroups of a CompoundTask that defines a Business Process

In the case of a CompoundTask defining a Business Process, this Class is used as a Callback object supplied by the invoker of the BusinessProcess to the factory operation mapped in section B.2.2. The Class is stereotyped <<BPCallback>>.

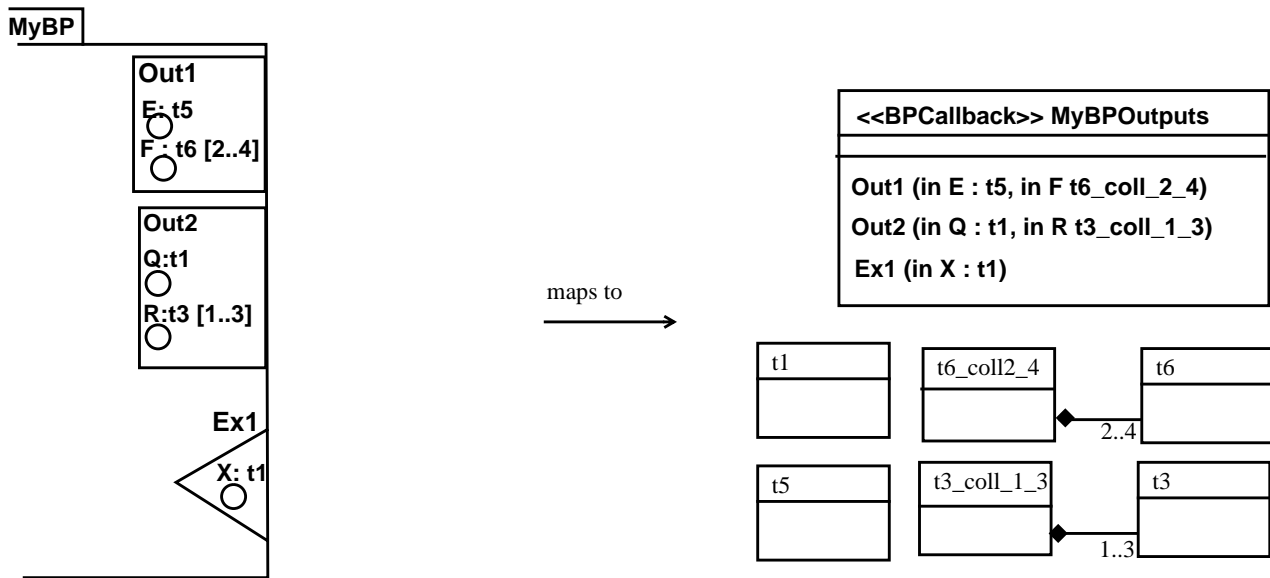


Figure B-8 Example mapping for the OutputGroups of a CompoundTask

## Mapping OutputGroups of an Activity that is performed by a BPRole

In the case of an Activity, the Class will be represented in a Collaboration by a ClassifierRole with this class as its base. It acts as a “portal” (or viewpoint interaction) between the Collaboration, representing the entity viewpoint of the system, and the Activity in the process viewpoint. It gives the modeler the hook needed to return the results of the execution of some feature of the object bound to the BPRole performing the Activity. The Class is stereotyped <<ActivityOutput>>.

Figure B-9 Example mapping for the OutputGroups of an Activity

### B.2.4 Mapping Asynchronous OutputGroups

Asynchronous OutputGroups are treated as Business Event Sinks attached to the Callback Entity representing the Outputs of the Business Process. See section B.3.

### B.2.5 Mapping Asynchronous InputGroups

Asynchronous InputGroups are treated as Business Event Sinks attached to the Entity representing the Business Process. See section B.3.

## B.2.6 Mapping a Business Process as an Entity

### Instances of the Business Process Class

The wrapper Class that represents the type of the Business Process will be instantiated by the generated Factory as an Object at runtime. It will be stereotyped as <<BusinessProcess>>. It will have attributes which encapsulate the values which were provided to it by its factory operation. It also inherits from a generic `BPManager` Class which represents the runtime infrastructure supported by the technology used to reify this model (for example the CORBA Workflow Facility interfaces). Each Object of the Service Wrapper type represents a single instance of the Business Process. It will also have Business Event Sinks attached to it representing the Compound Task's asynchronous InputGroups.

### Mapping Inputs to Attributes

The values provided by the creator of the Business Process instance to a factory operation corresponding to one of its InputGroups will be available as public attributes on the Wrapper Class. The attributes have their changeability metaattribute set to frozen. The name of each Input is prepended with the name of its containing InputGroup, and an underscore, to give a unique name for the Attribute, and its type is the same as the type of the parameter to which the value was passed.

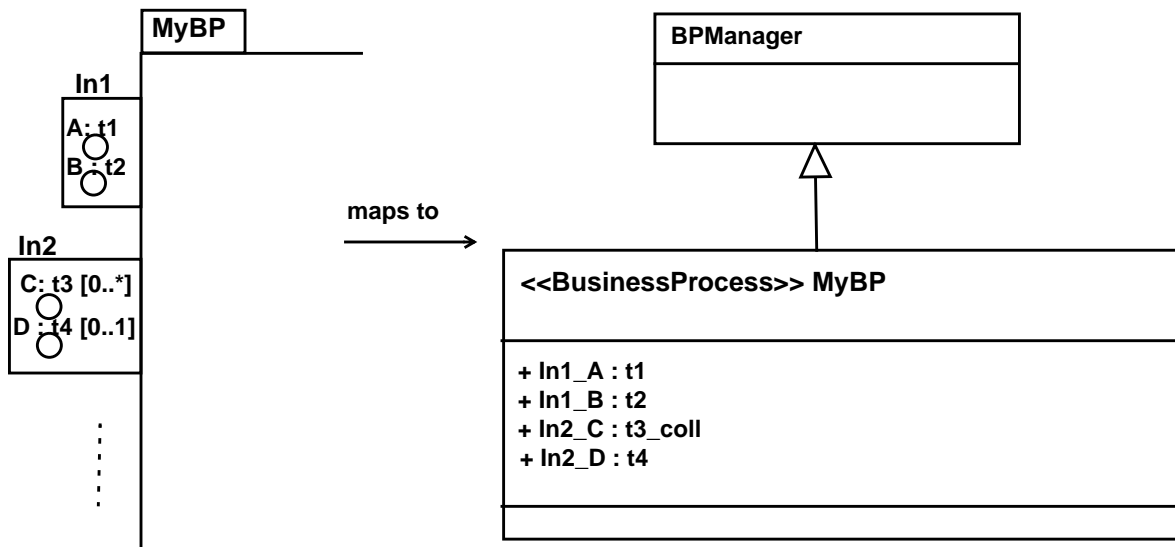


Figure B-10 Example mapping of the InputGroups of of a CompoundTask to a Business Process Instance

### *The Invocation Model for a Service Wrapper*

Three object implementations are required to instantiate and interact with a Business Process in the entity viewpoint.

1. The <<BPFactory>> object, implemented by the Business Process Infrastructure as a singleton. This object creates <<BusinessProcess>> object instances.
2. The <<BusinessProcess>> object itself is implemented by the EDOC infrastructure, and is an instance of the Class generated by mapping the synchronous InputGroups of the Business Process definition to operations. It will also have associated Objects implementing the Business Event Sinks that represent the Business Process's asynchronous InputGroups, as well as a link to the Callback object provided by the creator of the Business Process instance.
3. The <<BPCallback>> object is instantiated by the caller of the factory, and a reference to it must be passed as the final ("Callback") parameter to the factory Operations representing the synchronous InputGroups of the CompoundTask defining the business process.

### *B.2.7 Generation of an Activity from Features of an Entity*

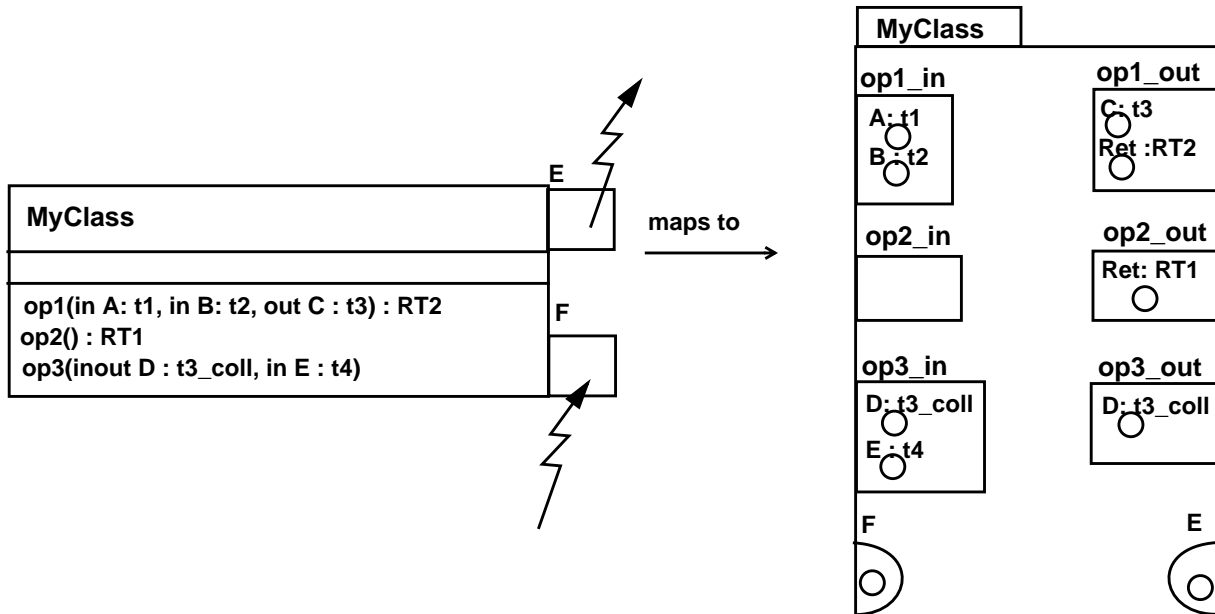
For each Operation or Method Feature owned by a Class, the parameters of the Feature are mapped to the contents of a synchronous InputGroup with the same name as the Feature and a synchronous OutputGroup with that name appended by "\_out".

- Each parameter that has kind "in" or "inout" is mapped to an Input of the same name in the InputGroup. Each Input's multiplicity is {1}.
- Each parameter that has kind "out", "inout" or "return" is mapped to an Output of the same name in the OutputGroup. Each Output's multiplicity is {1}.

For each Business Event Sink you wish to be exposed by the Entity an asynchronous InputGroup is created, with a single Input of the type of the Business Event Type. The Input's multiplicity is {1}. The InputGroup will have the same name as the Sink, and the Input's name is arbitrarily generated.

For each Business Event Source you wish to be exposed by the Entity an asynchronous OutputGroup is created, with a single Output of the type of the Business Event Type, and multiplicity of {1}. The OutputGroup will have the same name as the Sink, and the Output's name is arbitrarily generated.

## Example

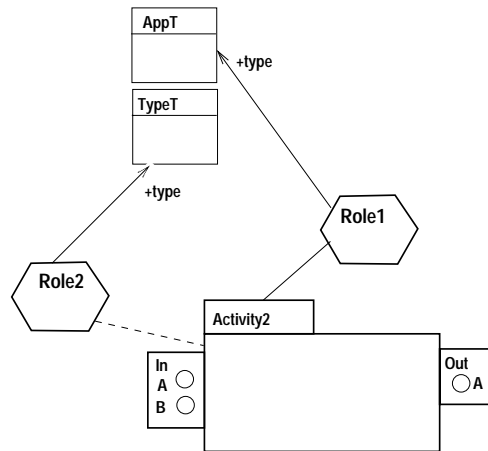


### B.2.8 Generation of Skeleton Collaboration from an Activity

For each activity that is performed by a BPRole, a skeleton Collaboration is generated that contains ClassifierRoles for each of the InputGroups, OutputGroups, and BPRoles associated with the Activity. An additional Actor ClassifierRole is shown to represent the delivery of values from the Business Process. The performer BPRole is given an unspecified Interaction with each of the artifact BPRoles used by the Activity. The Activity's InputGroup and OutputGroup ClassifierRoles are given unspecified Interactions with the performer BPRole's ClassifierRole. The modeler then chooses the Messages that represent method invocations and other feature usages in order to achieve the effect that the Activity represents in the Process Model. See the example in Figure B-11 on page 92.

## Input Portal Classifier Role Generation

A ClassifierRole is created that has the Class generated as per Section B.2.2 “Mapping Synchronous InputGroups” as its base, and all the generated Operations as availableFeatures. The ClassifierRole is given the name of the Classifier with “Role” appended.



generates skeleton

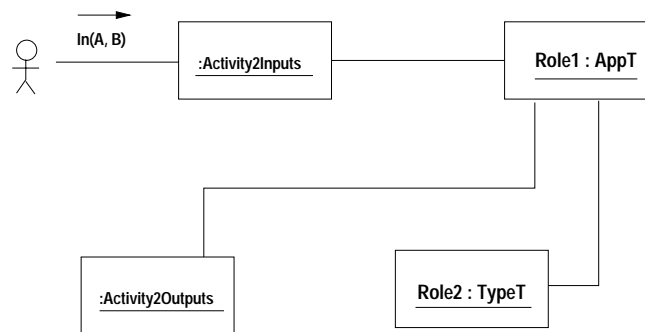


Figure B-11 Example Skeleton Collaboration mapping

## Output Portal ClassifierRole Generation

A ClassifierRole is created that has the class generated as per Section B.2.3 “Mapping Synchronous OutputGroups” as its base, and all the generated Operations as availableFeatures. The ClassifierRole is given the name of the Classifier with “Role” appended.

## Associated BPRole ClassifierRole Generation

Each Activity in a CompoundTask that is performed by a BPRole may have its interactions with the object bound to this BPRole defined by a Collaboration in which the Classes mapping the DataGroups of the Activity, as well as those that type the

BPRoles represented by ClassifierRoles. The Collaboration and these ClassifierRoles are generated as a skeleton, and the modeler may then choose exactly how to invoke the features of the Roles by adding Interactions and Messages to the skeleton.

### B.3 Mapping Asynchronous DataGroups to Event Transceivers

What does it mean to have DataGroups on Task Boxes representing asynchronous flows?

- It means that the Task plays the “source” role of a EventSource (Source for short) or the “sink” role of a EventSink (Sink for short).
- For Compound Tasks an asynchronous InputGroup represents a Source that is available to the nested ModelElements in the Compound Task which may have Sinks attached to them.
- For an Activity an asynchronous InputGroup represents a Sink that can subscribe to events from outside the Activity. It will have a matching Source in the Compound Task that defines the Activity, or the Entity that performs the Activity will have such a Source.
- During execution an Activity will hold at most one set of values in its asynchronous InputGroups until the Activity is enabled through one of its synchronous InputGroups. At this point the Compound Task that defines the Activity will be instantiated, or the Performer Role that performs the activity will be bound, and then these values will be the first to be conveyed to the Sink attached to the instantiated Task or bound Object. While the Activity is executing (until one of its synchronous OutputGroups is enabled) flows arriving at an asynchronous InputGroup will be passed through to the Sink inside the Activity each time the InputGroup is enabled.
- Using an asynchronous InputGroup as a correlator. Asynchronous InputGroups with only one Input will be satisfied as soon as the multiplicity of the Input is reached, and all values received will flow inwards. However, when there are multiple Inputs, each of these Inputs must be satisfied for the InputGroup as a whole to be passed inwards. This has the same OR semantics as synchronous InputGroups. That is, while the entire InputGroup is not yet satisfied, and more values flow to one Input than its multiplicity allows, unspecified values will be discarded to keep the Input at its specified multiplicity. For example, an InputGroup with Inputs A: X [1..1] and B: Y [2..4] which receives a value at Input A, and then a value at Input B, followed by another value at Input A will result in one of the values at A being discarded, because B is not yet satisfied.
- For a Compound Task an asynchronous InputGroup represents a Sink.
- For an Activity an asynchronous OutputGroup represents a Source. This allows Events coming from the Task instance it references to be propagated to other Activities at the same scope.

What does it mean for an Activity performed by a Performer Role to have an asynchronous InputGroup?

- It implies that the Collaboration representing the Objects which will actually do the work of the Activity will contain at least one component that can act as a Sink for the events of the type represented by the InputGroup. Note that if the InputGroup contains more than one Input then the Event Type

What does it mean for a BusinessProcess to have asynch DataGroups?

- This implies that from the Entity Viewpoint, the Process Entity representing the BP will emit/consume events corresponding to the types of DataElements of the asynchronous DataGroups.

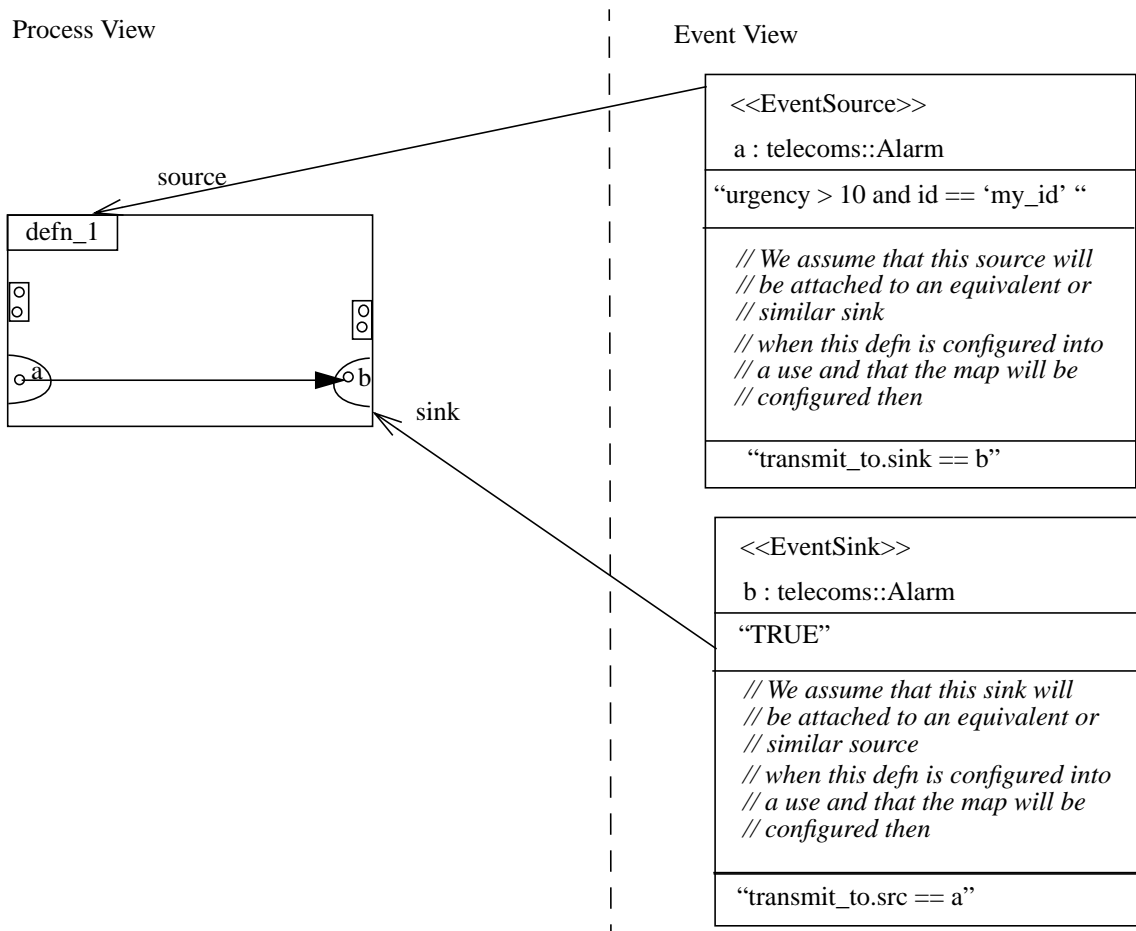
What additional modeling benefits come from adding asynch DataGroups to Tasks?

- It means that the subscription of a sink (asynch InputGroup) lasts for the active lifetime of the Task (i.e. from when one of its synchronous InputGroups being enabled starts the task running, to when one of its synchronous OutputGroups becoming enabled stops the Task from running). This is a powerful way to express the temporal and/or synchronization constraints over a subscription to a event type. From the Entity Model viewpoint only the intention to emit or consume events can be declared, but the times/conditions under which emission or consumption will take place cannot be specified without specifying the internals of the Entity in terms of state charts, sequence diagrams etc.

What does it mean to draw a flow from an asynchronous DataGroup to a synchronous DataGroup?

- The following subsections will explain the semantics for all well formed flows to and from the asynchronous DataGroups in terms of pairs of EventSource/EventSink with a “transmits\_to” association between them.

## B.3.1 Asynch InputGroup Flows to Asynch OutputGroup



### Description

This is a well formed but trivial flow that allows filtering of Events by placing subscriptions on DataGroups 'a' and/or 'b'. In the example we filter at 'a'. The

### Mapping

DataGroup 'a' maps to Source 'a', and DataGroup 'b' maps to Sink 'b'.

## B.3.2 Asynch InputGroup Flows to Asynch InputGroup

### Description

The heading is potentially misleading, as what is really happening is that an In Flow on a Task Definition (a Source) is sending events to an In Flow on a Task Use (a Sink), not to another Source, as the heading implies. In the figure below the InputGroup k

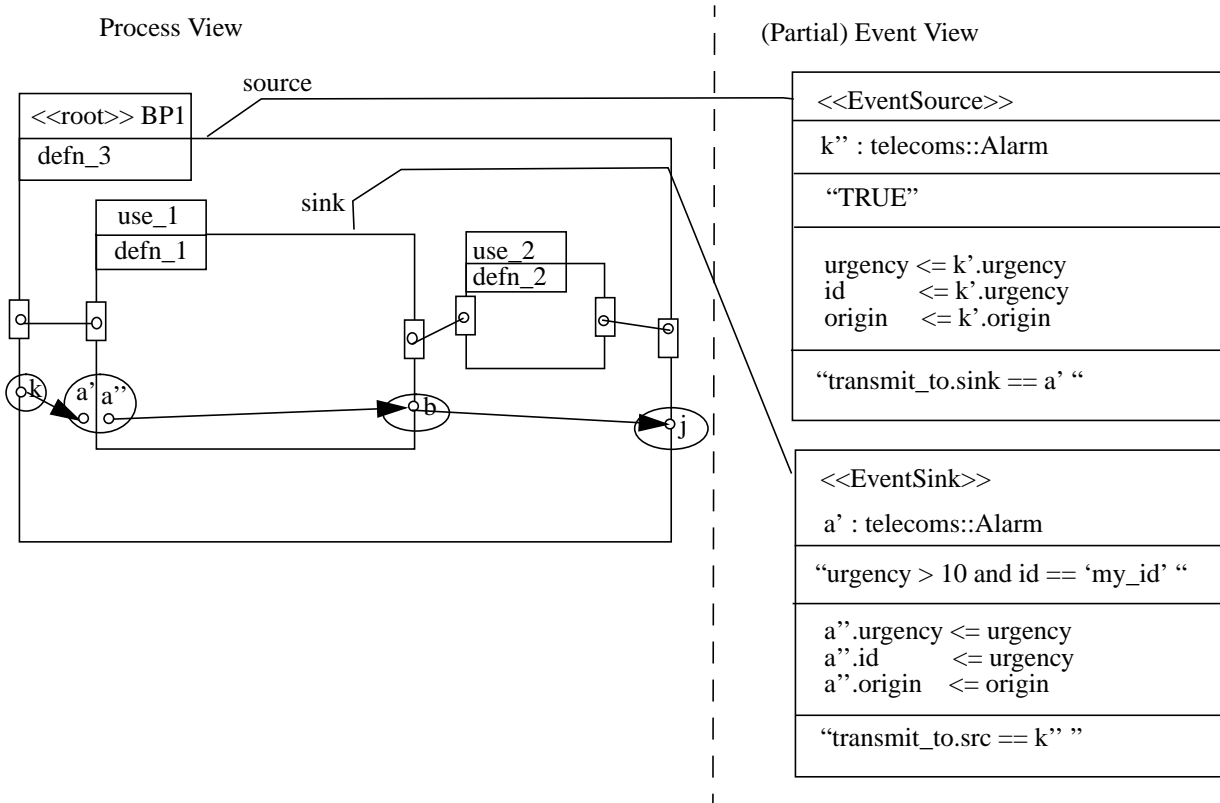
# Model to Model Mappings

really represents a Sink (k') on the Task Use (in this case a root Use, or Business Process) BP1, as well as a Source (k'') on the Task Definition defn\_3. This is expanded to demonstrate the point in the case of a, which is shown explicitly as a' and a''.

Let us consider a single flow from the example below, that from k to a, which is really a transmits\_to association between the Source k'' and the Sink a'.

### Mapping

Input k maps to Sink k' and Source k'', and Input a' maps to Sink a'.



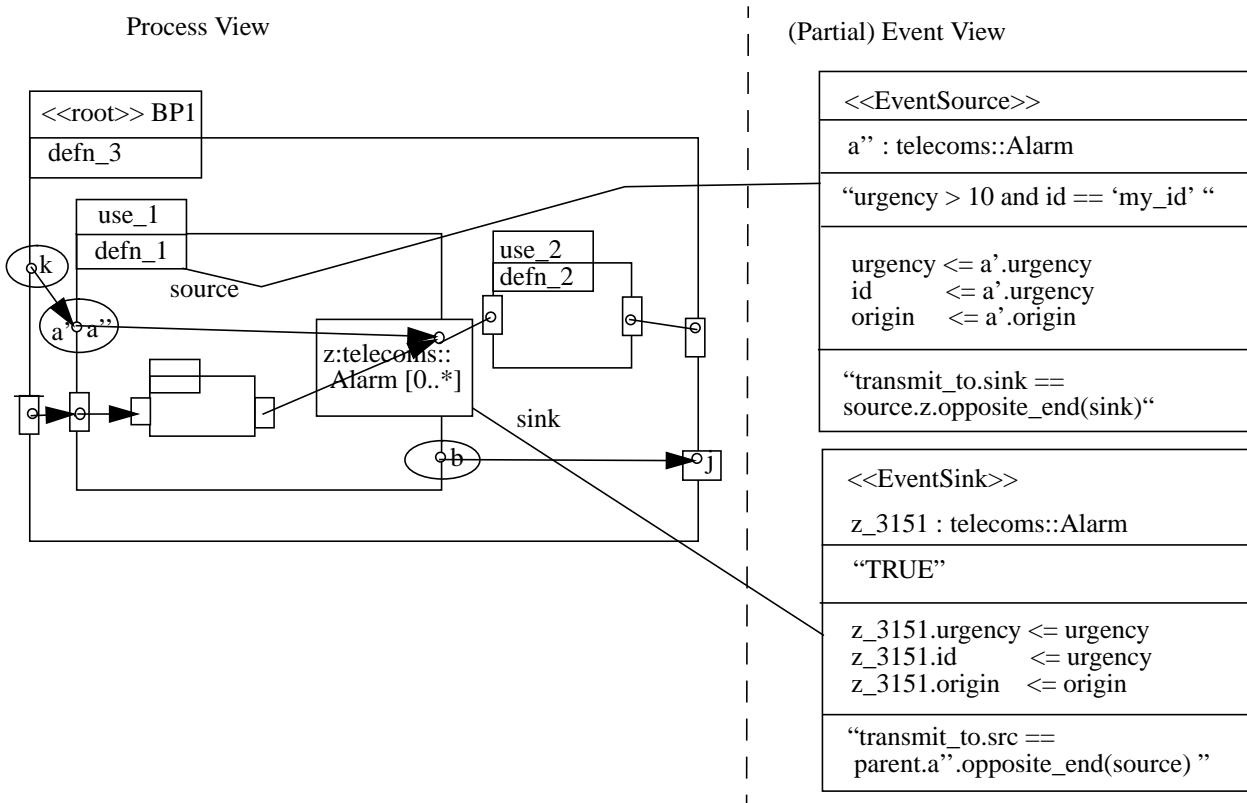
### B.3.3 Asynch Input Flows to Synch Output

#### Description

The events received through an asynch Input can be directed to a synchronized Output if that Output has the same type. In most useful cases this Output will have a multiplicity greater than 1, so that it can accumulate many incoming asynchronous events, and make them flow synchronously as a result to the next task once the OutputGroup containing that Output is enabled.

#### Mapping

Input a'' maps to Source a'' Output z gets an attached Sink z<generated\_id>.



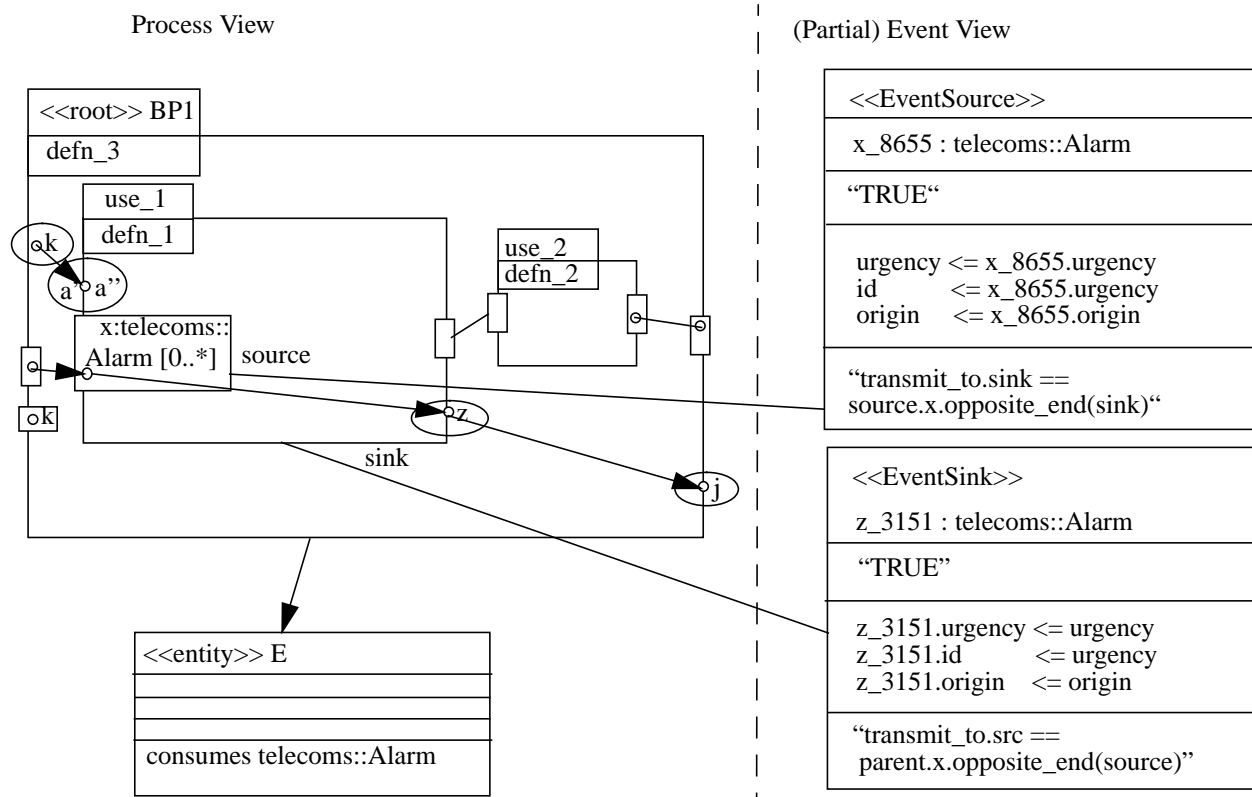
### B.3.4 Synch Input Flows to Asynch Out Port

#### Description

This kind of flow allows any Input which matches the rule of the Source attached to the Input AND the Sink representing the Asynch Output to be delivered as a batch of Events (or a single Event if the Input has multiplicity 1) to that Output. There they will be queued for delivery to the Sinks connected to its external Source.

#### Mapping

# Model to Model Mappings



# *Mapping to CORBA and Services Appendix C*

---

## *C.1 Introduction*

Models of enterprise applications created using the EDOC Profile specified in the main body of this document are at a high enough level of abstraction that several possible mappings to CORBA and CORBA services are possible. This appendix aims to provide plausible mappings of all model elements contained in the EDOC meta-model to at least one underlying technology.

### *C.1.1 Alternative Mappings*

In the case of some model elements there are obvious mapping alternatives given the richness of functionality now available within the OMA. This approach anticipates a submission to the future EDOC RFP4 which will propose an intermediate level in the mapping containing abstractions of the capabilities of various parts of the CORBA Core and CORBA services so that the most appropriate underlying implementation technology may be chosen. For example, the transmission of asynchronous messages between model elements may be implemented using CORBA Messaging, the DII, CORBA Events or Notification. Model elements requiring such transmissions could be mapped to a service-independent representation, for which the most appropriate actual service can be substituted at development time.

Given the requirement that the mappings for the EDOC model elements be non-normative, we have attempted only to demonstrate that multiple mappings are possible (even desirable) rather than attempting to give a framework for alternative mappings. This work is best left until a stable EDOC Profile is adopted and a normative mapping is requested. At this stage a proposal to support any plausible implementation style for enterprise applications will be more appropriate.

## C.1.2 Structure

This appendix is structured to mirror the structure of the main chapters of the submission, occasionally duplicating part of that structure where a coherent set of model elements have a mapping to alternative underlying part of the OMA.

Section C.2 gives a mapping for the Execution and Monitoring aspects of the Business Process Model, using the Workflow Management Facility. Sections C.3 and C.4 provide alternative mappings for the DataFlow aspects of the Business Process Model.

Section C.5 provides the mapping for the Business Event Model.

## C.2 Common Base Types for the Business Process Model

The Workflow Management Facility defines a number of interfaces for the execution, monitoring and meta-data query of what we have modeled as Activities, CompoundTasks, and Business Processes. These are used as a common basis for the alternative mappings of the Business Process Model.

### C.2.1 Task (*abstract*)

The Workflow Model presented in the Workflow Management Facility makes a distinction between Processes and Activities which parallels our Task subtypes CompoundTask and Activity respectively. Task is therefore mapped as `WorkflowModel::WfExecutionObject`, which is the abstract base type of Process and Activity.

In an implementation based on the Workflow Management Facility the responsibility for enacting an Activity is divided between a Workflow Engine which executes Activities definedBy CompoundTasks, and entities selected by performedBy BPRoles which execute Activities.

We defer the rest of the Task mapping to its derived model elements' mappings.

### C.2.2 BusinessProcess

A BusinessProcess is the implementation of a CompoundTask, and as such, it is implemented by a `WorkflowModel::WfProcessMgr` object, as defined in the mapping of CompoundTask.

### C.2.3 CompoundTask

CompoundTasks have both a type and an instance nature.

The type nature of a CompoundTask is mapped to a type manager, which in the Workflow Management facility are `WorkflowModel::WfProcessMgr` objects.

During execution the enabling of an Activity `definedBy` a CompoundTask causes an instance of that CompoundTask type to be created. This instance nature of CompoundTasks is mapped as a `WorkflowModel::WfProcess` object. The key attribute of the `WfProcess` must be an instance identifier. This identifier is used (as a parent Task Id) in the mappings of the DataGroups and DataFlows in the following sections.

When the `WfProcess` implementing the CompoundTask is run, it must also create instances of `WorkflowModel::WfActivity` for each Activity it directly contains.

### C.2.4 Activity

Activities are mapped to `WorkflowModel::WfActivity` objects. Through the mapping of the Activity's `InputGroups` and `uses` associations, it will be able to pass Input values and references to bound entities.

The complete mapping of an Activity depends on whether it has `performedBy` associations to `BPRoles` or a `definedBy` association to a CompoundTask; Activities that are `definedBy` a CompoundTask are mapped to objects that also implement the `WorkflowModel::WfRequester` interface.

#### *Associations*

##### *uses*

The `uses` association between Activity and `BPRole` is a way of defining access requirements of Activities to entities residing outside the process model. Each link of this association kind is mapped as the existence of a `NameValue` member of the `process_context` attribute of the `WfExecutionObject` which implements this Activity; the `_name` in the `NameValue` is given the `BPRole`'s name, and the `_value` is an object reference of the same type as the `BPRole`. At runtime the object referred to will be chosen (using the `type` association and the `find` and `factory` Expressions). See the mapping of `BPRole` in Section C.2.6 for details.

##### *performedBy*

The `performedBy` association specifies the `BPRole` which represents the behaviour to be executed by this Activity. The nominated `BPRole` may represent the interface to a person or group of people, or it may be a fully automated program that processes the Activity's Inputs and produces some Outputs. The association is implemented as a `WorkflowModel::WfAssignment`. The `BPRole` with which it is associated must support the type `WorkflowModel::WfResource`.

The `WfResource`'s `resource_key` and `resource_name` attributes may be used by the `BPRole` to locate and bind an entity of the appropriate application type to perform the Activity. Often this entity will represent a work list that will use the Activity's name, Inputs, and the `BPRoles` participating in `uses` associations with this Activity, to create a work item which is sent to a person, or group of people for processing.

##### *definedBy*

Activities that are defined by a CompoundTask are mapped to objects that also implement the WorkflowModel::WfRequester interface. Its DataGroups and DataFlow associations are mapped in Sections C.3 and C.4.

The association itself is implemented as an object reference to the WfProcessMgr instance representing the factory for the CompoundTask type. The Activity calls the ProcessMgr's create\_process() operation.

Once an instance has been obtained, the Inputs and BPRole bindings are provided through a ProcessData object referenced by the process\_context attribute. The appropriate map Expressions from the DataMaps are used to transform the input values before providing them to the ProcessData object.

### C.2.5 DataMap

Input values are provided to a WfProcess object (representing a CompoundTask) through its process\_context attribute. When initializing this object, the map Expressions from the appropriate DataMap belonging to the Activity that is defined by the CompoundTask are used to transform the input values and the results are stored in the process\_context attribute.

Similarly, Output values are made available via the results attribute of a WfActivity (representing the Activity that was defined by a CompoundTask). Again, the map Expressions from the appropriate DataMap are used to transform the output values and the results are stored in the results attribute.

### C.2.6 BPRole

A BPRole is mapped in CORBA as a set of object reference variables in use in some context. This is a novel modeling concept in the OMA, as specifications of clients of CORBA objects, and the binding process by which client code comes to refer to the "right" objects, has been impossible until now.

The BPRole concept recognizes that interface type compatibility is not sufficient to ensure that an object implementing the correct behavioural semantics is invoked by a client. BPRole is a kind of abstract behaviour, with both an interface type slot, and two kinds of criteria for selection of object instances to fill the role:

- Its find attribute - which allows the behaviour specification to express criteria by which objects that may fill the BPRole may be selected.
- Its factory attribute - which allows creation of objects which may then fill the BPRole.

#### *Binding*

The mapping for filling a BPRole is as follows. A simple BPRole may have many object reference types. For each BPRole, an instance of a business entity (a CORBA Object) must be located. The model elements provide a number of options to modelers to specify their binding constraints. Here are some of them:

The `find` expression of the `BPRole` may provide:

- a key for use with a factory/finder (type manager) in order to locate an appropriate object;
- an Interoperable Naming `iiopname` or `iioploc` URL which nominates a specific object;
- a Naming Context or hierarchy of Contexts which contain appropriate objects;
- a Trader Service Type and Constraint expression which will match appropriate objects in the Business Domain's Trader.

The `factory` expression of the `BPRole` may provide:

- a key for use with a factory/finder (type manager) in order to create an appropriate object;
- appropriate parameters for passing to a Factory to construct a new object.

All of these options are available as mechanisms for Tool Vendors to allow modelers to expose the requirements for the objects filling their `BPRoles` in the Model, and allow code to be generated that satisfies these requirements, rather than having programmers write magic bootstrapping code.

### C.3 Notification-based Mapping for the Business Process Model

In addition to the base interfaces defined in Section C.2, the following implementations must be provided for the elements in a Business Process Model. We envisage that they will eventually be implemented as CORBA Components with a separate facet for each of the interfaces required to be supported. However, in the absence of a Component-based ORB, they will usually be implemented by a number of cooperating servants in the same address space that each expose one or more object references. The desire to avoid name mangling of element names from the model to avoid operation and attribute name clashes means that multiple inheritance is impossible in some cases.

In this mapping, a number of model elements are subsumed into behaviours of the mappings of other model elements. The general approach is that `DataFlows` between `Tasks` are implemented as Structured Event transmissions between `Tasks`. As any `Input` or `Output` may be a source or a sink for a `DataFlow`, all the mapping is done at the level of the abstract model elements `DataGroup` and `DataElement`. The conditions under which `DataFlows` are transmitted, and the semantics of the arrival of a `DataFlow` are well defined in the Business Process Model, and this mapping (as well as the Interface-based mapping in Section C.4) concentrates only on the method of transmission of `DataFlows`.

### C.3.1 Task (abstract)

#### *DataFlow source*

Each Task which directly contains DataFlow sources must implement the CosNotifyComm::StructuredPushSupplier interface and connect to a Notification Channel created for the use of this BusinessProcess instance. The mapping of a DataFlow source's DataElement (Section C.3.2) prescribes the events types to be emitted by the Task to represent the DataFlows that these DataElements are sources for.

#### *DataFlow sink*

Each Task which directly contains DataFlow sinks must implement the CosNotifyComm::StructuredPushConsumer interface and connect to a Notification Channel created for the use of this BusinessProcess instance. It must create and attach a Filter to the ProxySupplier of the Event Channel to which it is attached. The mappings of a DataFlow sink's DataElement (Section C.3.2) provides constraints to be added to the Filter to ensure that the events representing DataFlows will be consumed at these sink elements.

### C.3.2 DataElement (abstract)

#### *DataFlow source*

Any DataElement that is the source of a DataFlow will create and transmit a Structured Event of the following type using the Event Channel to which its containing Task is connected.

**domain = "EDOC"**  
**name = "data\_flow"**  
**properties =**

**flow\_id : string // contains the data flow's fully qualified name**  
**source : string // contains <DataElements's fully qualified Name>**  
**parent : string // contains <Containing Task Instance ID>**  
**payload : any // contains the value(s) of the DataElement**

Note that the **flow\_id** for an ordinary flow is fixed in the model, and in the event it is scoped by the **source** property.

#### *DataFlow sink*

Any DataElement that is the sink of a DataFlow must create a subscription to add to its containing Task's Filter which subscribes to the event type EDOC/data\_flow, and has a constraint which selects events with the right flow\_id, and source name. The parent property must also be the same as the parent of the Task containing this DataElement.

## C.3.3 CompoundTask

### ExceptionGroup handling

A CompoundTask has responsibilities in addition to those of other Tasks. Each CompoundTask must have a subscription to Events of the EDOC/exception type. The only constraint is that the exception event was emitted by a Task contained directly by this CompoundTask. This can be expressed as:

**“parent == <My Instance Id>”**

Upon receipt of such an event the CompoundTask must terminate all its contained Activities, and then pass the payload of the event to the Outputs in its `system ExceptionGroup`.

## C.3.4 ExceptionGroup

ExceptionGroups are special OutputGroups that indicate a failure in the Task that contains them. An Activity's ExceptionGroup may either be *handled*, or the data values from its Outputs must be propagated to its containing CompoundTask's `system ExceptionGroup`.

If an ExceptionGroup is unhandled, that is its Outputs are not sources for any DataFlows, then the following event type, which will be subscribed to by the containing CompoundTask, must be emitted when the ExceptionGroup is enabled:

```
domain = "EDOC"  
name = "exception"  
properties =  
  source : string // contains <ExceptionGroup's fully qualified Name>  
  parent : string // contains <Parent's Instance ID>  
  payload : CosNotification::PropertySeq // contains name/value pairs  
    // corresponding to its Outputs
```

If an ExceptionGroup is handled (any of its Outputs are the source of a DataFlow), then it only emits ordinary EDOC/data\_flow events as specified in Section C.3.2.

## C.4 Interface-based Mapping for the Business Process Model

This section is an alternative to the mappings provided in Section C.3, but it still requires the mappings in Section C.2 as a basis.

The approach taken in this mapping is to implement all DataFlows as invocations on operations representing DataFlow sinks. The source of the DataFlow therefore is represented as an object reference to the DataGroups containing these sink points. To facilitate design (and mapped implementation) re-use, every potential DataFlow sink (i.e. every DataGroup) will be represented as an interface, so that the only runtime configuration required is the finding of object references to the DataGroups which contain the sinks to the actual DataFlows in the Business Process Model.

## C.4.1 Task (abstract)

A Task is an instance of a Component, or server supporting an object reference for the Workflow interfaces defined in Section C.2, and an object reference for each DataGroup contained by the Task.

### Containment

A Task may need to be aware of its parent:

```
module EDOC {  
    interface TaskNavigation {  
        TaskNavigation my_container();  
    };  
};
```

The role of a Task in this mapping is to provide a NameSpace for the objects it contains. In a CORBA interface mapping this is done via modules:

```
module <Task Name> {  
  
    interface <Task Name>Navigation : EDOC::TaskNavigation;  
  
    // CompoundTask interface goes here if this is a CompoundTask  
  
    // interface definitions for contained DataGroups go here  
  
    // possible statically generated DataFlow sources interface goes here  
  
    // modules representing contained Activities go here  
  
};
```

### DataFlow source

All Tasks support a generic interface that allows their Containers to provide them with the object references that they require to send out their DataFlows.

```
module EDOC {
  interface TaskDataFlowSource {
    add_data_flow_sink(
      in string source_data_element_name,
      in Object sink_obj_ref,
      in string sink_data_element_name);
  };
};
```

This allows a DataFlow to be described in terms of the source name (of the form DataGroupName::DataElementName), and the object which defines its sink DataGroup, as well as the name of the method to be called on that object. As defined in Section C.4.3, the method to be invoked is named the same as the sink DataElement, and it always has a single parameter called “values”, which is of the same type as the source DataElement.

In addition, the mapping may generate static interfaces of the form:

```
interface <TaskName>Sources {
  add_<DataFlowName>_sink(
    in <CompoundTask Module>::<sink's DataGroup Name> sink);
  // etc...
};
```

There will be an operation per DataFlow for which this Task is a source. The generated code will be able to statically invoke the right operation on the sink object reference passed in to each of these operations.

## C.4.2 DataGroup (abstract)

### *DataElement Container*

A DataGroup contains a fixed (possibly empty) set of DataElements, each with a unique name. The following interface is generated to map the DataGroup:

```
interface <DataGroup Name> {
  // Contained DataElement Mappings go here
};
```

There is no distinction in the interfaces between synchronous and asynchronous DataGroups; the objects implementing the interfaces must provide the appropriate semantics.

### C.4.3 DataElement (abstract)

Each DataElement is represented as an operation of the form:

```
void <DataElement Name> ( in values <type attribute mapping>);
```

The type of the 'values' parameter should be a collection type (i.e., a sequence) to support DataElement multiplicities other than {1,1}.

### C.4.4 CompoundTask

#### *DataFlow Container*

A CompoundTask contains all the DataFlows that connect the Activities which it contains. This means that the CompoundTask is responsible for passing object references of the DataGroups which are sinks of DataFlows to the Activity containing the DataGroups that are the sources of these DataFlows.

It may do this by making calls to the generic TaskSource interface (assuming that Tasks making calls can use the DII), or to the statically typed generated <TaskName>Sources interfaces that may be generated after this Task's context in the Model is known.

#### *Exception Catcher*

All CompoundTasks support an interface derived from:

```
module EDOC {  
  
    interface CompoundTask {  
        void system_exception(  
            in payload CosNotification::PropertySeq);  
    };  
};
```

The interface is defined as:

```
interface <Task Name>Compound : EDOC::CompoundTask;
```

The payload contains a Property for each Output in the Exception.

### C.4.5 ExceptionGroup

Unhandled ExceptionGroups must call the `system_exception()` operation on their Container's CompoundTask interface. Handled ExceptionGroups (ones with DataFlows proceeding from their Outputs) behave the same as ordinary OutputGroups.

## C.4.6 BusinessProcess

### Containment

In this mapping a Business Process indicates which Task Containment level (indicated by its realizes association with a CompoundTask) is significant enough to give an outer-level module scope to.

```
module <BusinessProcess Name> {  
  
    // realized CompoundTask declarations go here  
  
};
```

## C.5 The Business Event Model

### C.5.1 Event Type

An Event's type is mapped to a Notification Service Event Type Repository entry.

#### Attributes

`domain_name`: becomes the name of the package containing the EDOC Model.

`type_name`: becomes the name of the Classifier indicated by the `type` association.

All attributes of the Event's type are mapped to Properties contained by the Repository EventType.

### C.5.2 EventTransceiver (abstract)

EventTransceiver is mapped to an abstract IDL interface for use in its derived types:

```
module EDOC {
  interface EventTransceiver {

    readonly attribute exposureRule string;

    readonly attribute type Any;

    readonly attribute map StringSeq;
  };
};
```

### C.5.3 EventSource

#### *Interface*

EventSource is mapped to an implementation of the following interface:

```
module EDOC {
  interface EventSource : EventTransceiver,
    CosNotifyComm::StructuredPushSupplier {

    void action ( in string action_name, in any context);

    void transition(
      in string transition_name,
      in string from_state,
      in string to_state,
      in any context);
  };
};
```

#### *Implementation*

The implementation of the EventSource must be a facet of a component implementing the `source` ModelElement, or have a servant in the same address space as the implementation of `source`.

The implementation will embed the Event `exposureRule` in the model, and generate code to provide appropriate pointers and object references to the names in the `exposureRule` representing values of the `source` Element. When a rule evaluates to true then a StructuredEvent of the type mapped by the `type` attribute of the EventSource is created.

The evaluation of the EventSource's `exposureRule` may be delegated to a Filter on the Proxy in the Notification Service to which it is connected. This requires that the BooleanExpression language chosen in the model can be translated to the Notification Service Constraint Language, and that the attributes that the rule evaluates are present as mapped properties in the generated event. In this case, the invocation of the

`action()` or `transition()` operations will always generate an event when the source is active, and the rule will evaluate as a filter which blocks the transmission of the event at the proxy if the rule is not satisfied.

If the `map` attribute of the `EventSource` is empty, then a Structured Event variable declaration and a “fill in code here” comment is generated so that a programmer using the generated code can populate the values of the `StructuredEvent`.

If the `map` is non-empty, then the code generated must provide appropriate local variable pointers and object references to the model elements mentioned in each of the `map`'s Expressions. This requires that the Expression language chosen in the model can be interpreted to obtain current values of the appropriate objects and then copy them into the corresponding `StructuredEvent` property's value members.

Finally, the event should be pushed to the Notification Service. (See the *Configuration* sub-section below for details of connection to the Notification Service.)

Implementations may choose to optimize the generation of events by implementing the `NotifySubscribe` interface, and checking for subscriptions to the event type being produced. As the specification of `EventSinks` below requires the use of the event type in a `Filter`, the Source can rely on the `EventType` being available to it via calls from the Proxy to its `subscription_change()` method.

### *Configuration*

The `extent` attribute of the `EventSource` is used to determine which Notification Channel is to be used to broadcast events that are generated. The standard values for `extent` are mapped as follows:

- “global” - The EDOC application as a whole will need a bootstrap parameter containing an object reference for an event channel connecting this application to others that wish to see events that it publishes globally. Alternatively the default channel for such events should be that obtained by calling `resolve_initial_references()` on the ORB with “NotificationService” as its parameter. Then the default channel (number 0) should be chosen, and the default `SupplierAdmin` used to obtain a Proxy.
- “application” - All EDOC applications using Business Events should have some bootstrap code that creates a new channel for internal events. This channel's reference should be stored in a standard place in the Naming Service for access by all `EventSources` and `EventSinks` with “application” extent.
- “direct\_only” - A new channel is created for each `EventSource`, and this channel's reference is obtained only by the `EventSinks` that are associated with it by the `transmission` association in the model.

The `EventSource` should use the `SupplierAdmin` interface to obtain a `ProxyStructuredPushConsumer` from the channel, and pass its own object reference to the proxy's `connect_structured_push_supplier()` method. Before sending events the implementation should offer the event type using the proxy's `offer_change()` method.

## C.5.4 EventSink

### Interface

Business Event Sink is mapped to an implementation of the following interface:

```
module EDOC {  
    interface BusinessEventSink : BusinessEventTransceiver,  
        CosNotifyComm::StructuredPushConsumer{};  
};
```

### Implementation

The implementation of the EventSink must be a facet of a component implementing the sink ModelElement, or have a Servant in the same address space as the implementation of sink.

The implementation may embed the Event exposureRule and generate code to provide appropriate pointers and object references to the names in the rule representing values of the sink Element. The rule is evaluated when the Notification Service calls the push\_structed\_event ( ) method on the sink. When it evaluates to true, the maps are applied. If the map attribute of the EventSink is empty, then a “fill in code here” comment is generated so that a programmer using the generated code can use the properties in the StructuredEvent.

If the map is non-empty, then the code generated must provide appropriate local variable pointers and object references to the model elements mentioned in each of the map’s Expressions. This requires that the Expression language chosen in the model can be interpreted to copy values from the StructuredEvent property’s value members to appropriate objects.

The evaluation of the EventSink’s rule may be delegated to a Filter on the Proxy in the Notification Service to which it is connected. This requires that the BooleanExpression language chosen in the model can be translated to the Notification Service Constraint Language and that all the variables in the rule refer to event properties, and not to model elements.

### Configuration

The extent attribute of the EventSink is used to determine which Notification Channel is to be used to subscribe to events that are generated. The standard values for extent are mapped as follows:

- “global” - The EDOC application as a whole will need a bootstrap parameter containing an object reference for an event channel connecting this application to others that wish to see events that it publishes globally. Alternatively the default channel for such events should be that obtained by calling resolve\_initial\_references on the ORB with “NotificationService” as it’s parameter. Then the default channel (number 0) should be chosen, and the default ConsumerAdmin used to obtain a Proxy.

- “application” - All EDOC applications using Business Events should have some bootstrap code that creates a new channel for internal events. This channel’s reference should be stored in a standard place in the Naming Service for access by all EventSources and EventSinks with “application” extent.
- “direct\_only” - A new channel is created for each EventSource, and this channel is subscribed to only by the EventSinks that are associated with it by the transmit\_to association in the model. A reference to this channel is shared using a name in the Naming Service.

The EventSink should use the ConsumerAdmin interface to obtain a ProxyStructuredPushSupplier from the channel, and pass its own object reference to the proxy’s connect\_structured\_push\_consumer() method. The EventSink must then create a Filter and add to it a ConstraintExp that subscribes to events of the EventType it requires.

The evaluation of the EventSink’s exposureRule may be delegated to a constraint expression string in the ConstraintExp used at the Filter on the Proxy to select events of the right type. This can be done if the BooleanExpression language chosen in the model can be translated to the Notification Service Constraint Language, and the variables that the exposureRule evaluates are all properties in the received event.

### C.5.5 Mapping Event Aspects of Specific Elements

The simple interface EDOC::EventSource contains two operations which are used to trigger the generation of an event based in its rule and mapping specification. It is assumed that the implementation of the EventSource is in the same address-space as the ModelElement to which it is attached, and therefore that in most cases no information need be transmitted along with the action or transition being indicated by the calling of these operations, as this is available as local data. However, in certain situations an encapsulation of some state is required in case some other agent modifies the thing causing the action before its relevant properties can be mapped into the event. In this case the context parameter of the action() or transition() method is used.

When the action operation is invoked, this causes the rule variable “action” in the rule to be given the value of the action\_name parameter.

When the transition operation is invoked, the “action” variable in the rule is given the value “transition”; the rule variable “transition” takes the value of the transition\_name parameter, and the rule variables “from\_state” and “to\_state” take the values of the corresponding parameters.

#### *Task*

A Task has an associated finite state machine (see Figure 2-2 on page 24). Each transition in the state machine causes the transition operation of all associated EventSinks to be called with the appropriate parameters.

The Task also contains a number of InputGroups and OutputGroups, and the Inputs and Outputs of these DataGroups can be named by using the form “Input-Group-name::Input-name”. These names can be used in `exposureRule` Expressions that expect the type of the Input or Output, or may be compared with the special value `NULL`. In addition, the Inputs and Outputs may be referred to in `map` Expressions, and when nominated at an EventSink the assignment of values into the Input or Output is equivalent to such values arriving from a DataFlow.

### *DataGroup*

A DataGroup’s actions (as specified in Section 3.5.3) are ‘satisfy’ and ‘enable’. These are indicated by invoking the EventSink’s `action()` method with the appropriate `action_name`.

The names of the Inputs or Outputs within this DataGroup may be used in `exposureRules` and they may be referenced by `map` Expressions, and, as in the case of Task, the assignment to one of these is equivalent to its arrival via a DataFlow.

### *DataElement*

A DataElement’s only action is to be assigned values, usually via an incoming DataFlow. The `action()` method is called on the EventSource with the `action_name` parameter ‘assign’. Along with the DataElement’s attributes (see DataElement and its subclasses in the Business Process Model in Figure 2-1 on page 23), the values assigned to the DataElement are available for use in the `exposureRule` for emission using the rule variable “values”.

### *DataFlow*

A DataFlow’s only action is named ‘transmit’, and it uses the `action()` method to alert the EventSource that it is about to send values.

### *BPRole*

The actions of interest that occur in BPRoles are as follows:

A BPRole’s action of interest is binding. The `action()` method is called on the EventSource with the `action_name` parameter ‘bind’.

*D.1 EDOC Model Modeling concepts and Activity Diagram notation*

This section shows how Activity Diagrams can be used to depict EDOC models as a transitional measure so that existing UML tools may be used immediately. Examples of each of the EDOC notations are provided along with the corresponding Activity Diagram depiction.

An EDOC Activity that is a leaf Task in some Business Process model and having synchronous DataGroups can be represented with the UML ActionState accompanied with several UML ObjectFlow states, each of which corresponds to an EDOC Data Element (i.e. an Input or an Output). Figure D-1 shows a simple case with one such EDOC Activity having one Input and one Output.

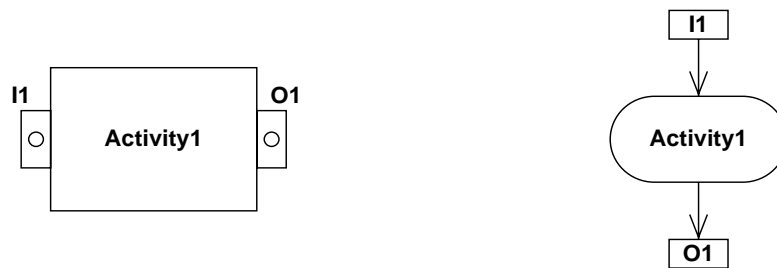


Figure D-1 Simple EDOC Activity and corresponding Activity Diagram notation.

# Activity Diagram Notation

In cases of an EDOC Activity with more than one Input (or Output) in a synchronous InputGroup (or OutputGroup) the AND semantics of Inputs (or Outputs) belonging to one InputGroup (or OutputGroup) can be represented using *join* kind of UML PseudoState, as shown in Figure D-2. Similarly the AND semantics of an OutputGroup can be shown by using a *fork* PseudoState.

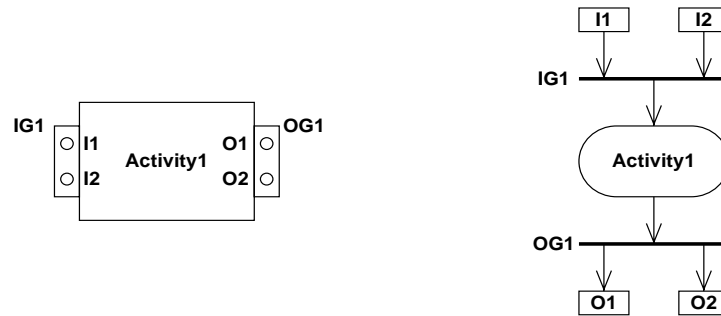


Figure D-2 Use of fork and join to capture DataGroup concepts.

Further, in cases where there are more than one InputGroups (or OutputGroups) one can draw more than one entries into (or exits from) an ActionState, corresponding to these multiple DataGroups (as shown in Figure D-3).

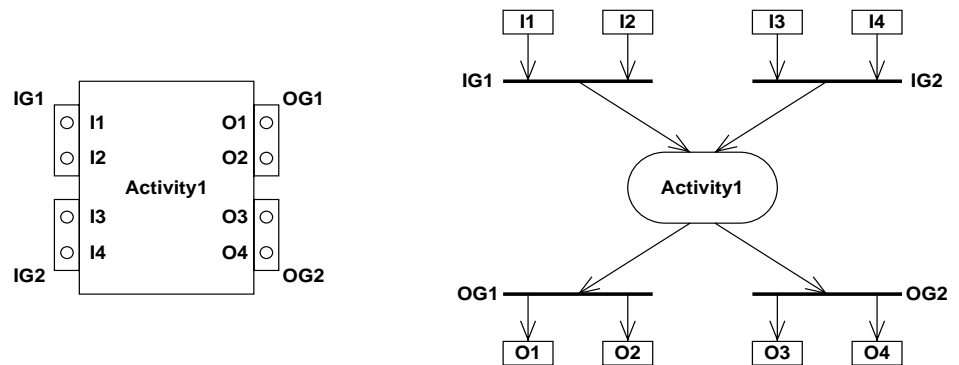


Figure D-3 Multiple DataGroups lead to multiple forks/joins.

In case of an EDOC Activity which is defined by a CompoundTask, the instantiation of this EDOC Activity can be represented with UML Initial State, followed by transitions into zero or more UML ObjectFlow states that correspond to CompoundTask's DataGroup specifications, in a similar manner as was case with the leaf kind of EDOC Activity as discussed above. This case is shown in Figure D-4.

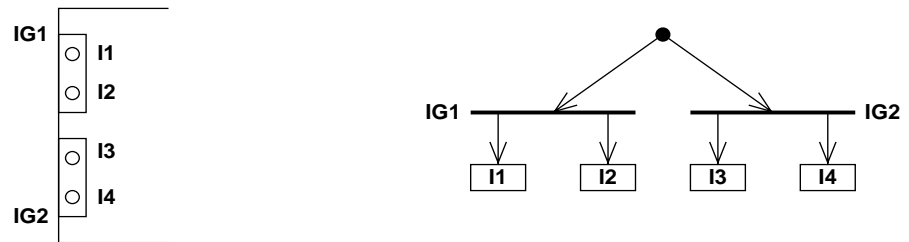


Figure D-4 A CompoundTask's InputGroups are represented as above.

We note that current tools limit drawing direct transition from an Initial Pseudo-state into ObjectFlow states and thus we recommend the use a dummy ActionState to deal with this limitation.

Similarly, for the completion of an EDOC Activity defined by a CompoundTask, we use FinalState preceded by those ObjectFlows that correspond to CompoundTask's Outputs, in a similar manner as with its Inputs. This is shown in Figure D-5. The tools limitation also require another dummy ActionState to be introduced, immediately preceding the FinalState.

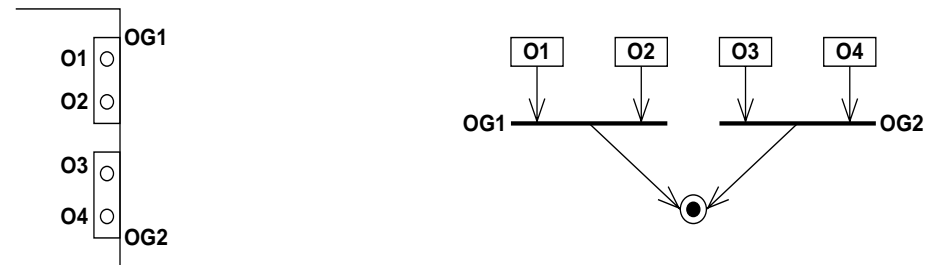


Figure D-5 A CompoundTask's OutputGroups are drawn as shown.

# Activity Diagram Notation

Asynchronous InputGroups/OutputGroups may be depicted using Control Icons for Signal sending and receiving. This is shown in Figure D-6.

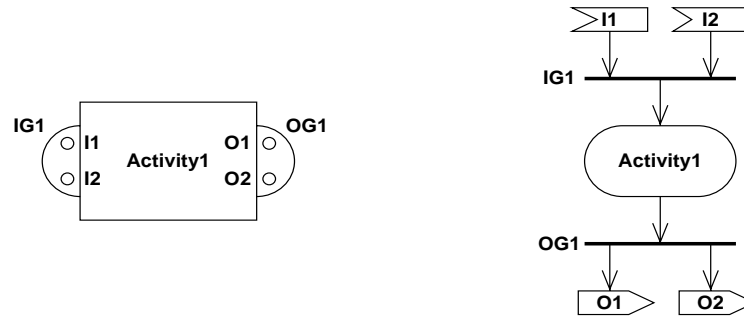


Figure D-6 Asynchronous DataGroups

ExceptionGroups are depicted in Activity Diagrams in the same manner as OutputGroups but with a stereotype tag <<exception>> attached to the fork PseudoState. An example of this is given in Figure D-7.

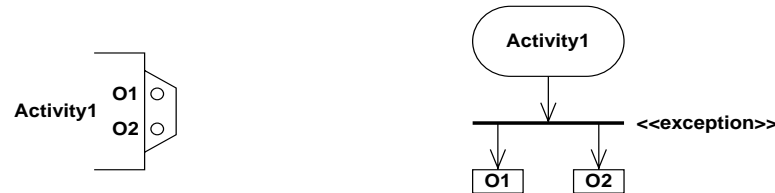


Figure D-7 An ExceptionGroup is depicted as a stereotyped fork.

Finally, the EDOC Compound Activity can be depicted using the UML Sub-Activity States, as shown in Figure D-8.

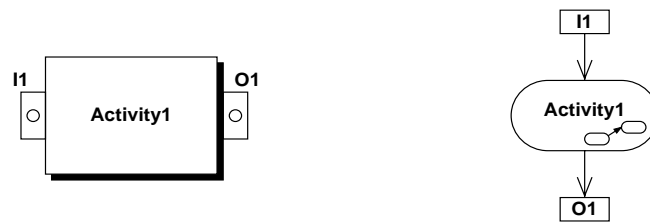


Figure D-8 Depicting an Activity defined by a CompoundTask.

Finally, in Figure D-9 we have an example of a complete CompoundTask and the corresponding depiction as an Activity Diagram in Figure D-10.

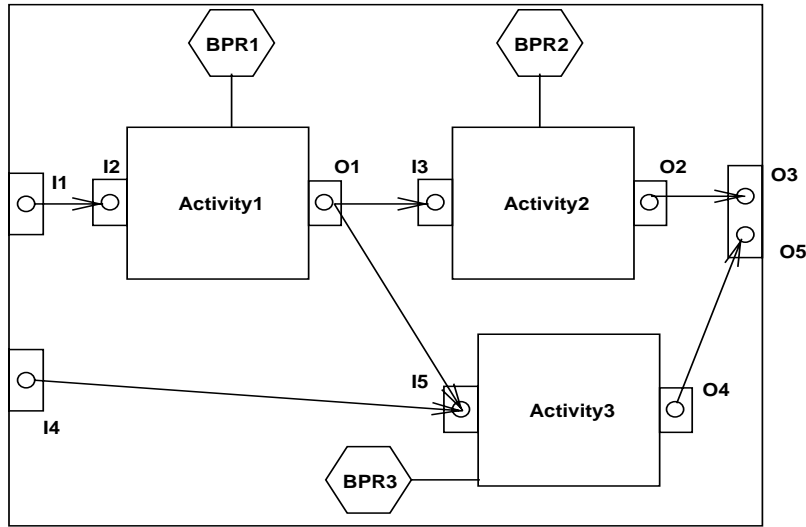


Figure D-9 CompoundTask example in EDOC notation.

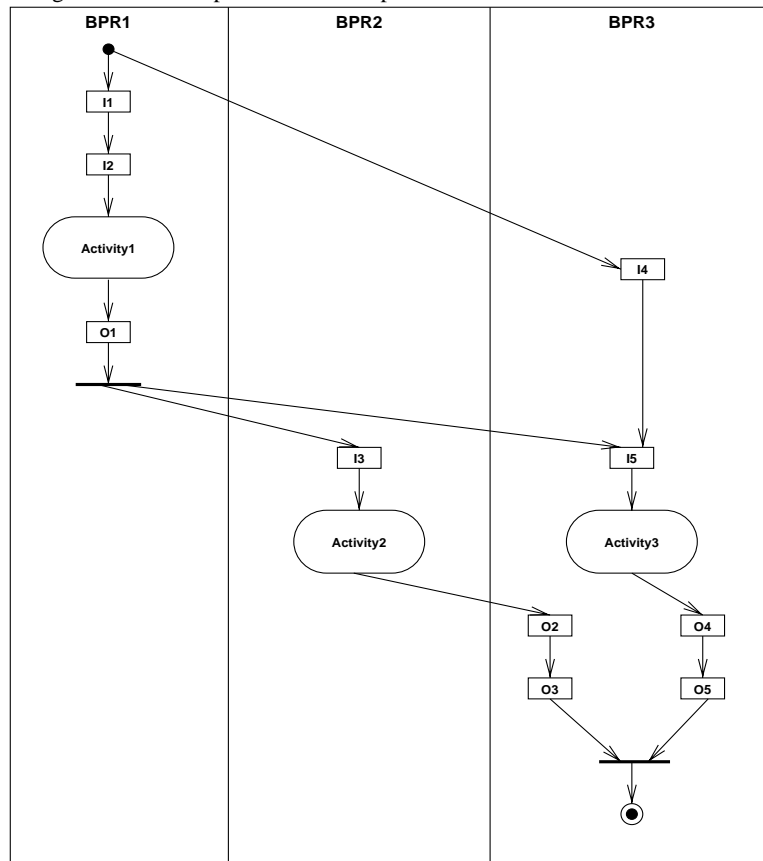


Figure D-10 Activity Diagram depiction of CompoundTask from Figure D-9.



### *E.1 Other Relevant Documents*

Document ad/2001-01-02 is a zip file containing the EDOC models in the MODL language, generated IDL interfaces, and an XMI DTD for the models.

These models have been verified to be MOF conformant using the dMOF product from DSTC. These tools were also used to generate the IDL and XMI DTD.

