**Date**: July 2007

# Business Process Definition MetaModel (BPDM), Beta 1

*OMG Adopted Specification*

OMG Document Number: dtc/07-07-01
URL: http://www.omg.org/cgi-bin/doc?dtc/2007-07-01
Associated Schema Files: http://www.omg.org/spec/BPDM/1.0/

This OMG document replaces the submission document (bmi/2007-03-01, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to *issues@omg.org* by September 14, 2007.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on December 21, 2007. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

# OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

# Table of Contents

# Preface

## About the Object Management Group

### OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

## OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

*http://www.omg.org/technology/documents/spec_catalog.htm*

Specifications within the Catalog are organized by the following categories:

### OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications

### OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

### Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

## Typographical Conventions

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

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

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

## Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/technology/agreement.htm.

# 1 Scope

The “Business Process Definition Metamodel” (BPDM) is a framework for understanding and specifying the processes of an organization or community. Business processes have been at the heart of business and technology improvement under the guise of many terms and methodologies, such as: Business Process Engineering or Re-Engineering, Business Process Management, Business Process Execution, Total Quality Management, Process Improvement, Business Process Modeling and Workflow. Similar and related concepts such as Service Oriented Architectures, Enterprise Application Integration, Flowcharts, Data Flows, Activity Diagrams, Role/Collaboration Modeling, and Modeling and Simulation serve to enable and describe business processes.

This heritage of process related approaches has provided substantial benefit to public and private institutions and is one of the factors that has allowed the modern enterprise to grow and prosper. This same heritage has also caused some confusion in how these various approaches and solutions do or do not work together and how to leverage them for a coherent and integrated solution. As of now there is a substantial asset of business process descriptions, notations, implementations and machinery but many of these are islands - islands of a particular technology, methodology or notation.

BPDM provides the capability to represent and model business processes independent of notation or methodology, thus bringing these different approaches together into a cohesive capability. This is done using a “meta model” - a model of how to describe business processes - a kind of shared vocabulary of process with well defined connections between terms and concepts. This meta model captures the meaning behind the notations and technologies in a way that can help integrate them and leverage existing assets and new designs. The meta model behind BPDM uses the OMG “Meta Object Facility” (MOF) standard to capture business processes in this very general way and to provide an XML syntax for storing and transferring business process models between tools and infrastructures. Various tools, methods and technologies can then map their way to view, understand and implement processes to and through BPDM.

To achieve this goal, BPDM supports two fundamental and complementary views of process - “Orchestration” and “Choreography.”

- Orchestration concepts in BPDM are represented through sequences of "Activities" that produce results with branching and synchronization. Orchestration is typically represented as flow charts, activity diagrams, swim lanes or similar notations of one task or activity following another. The orchestration of processes describes what happens and when in order to better manage a process under the authority of some entity.
- Choreography describes how semi-independent and collaborating entities work together in a process, each of which may have their own internal processes. Choreography captures the interactions of roles with well defined responsibilities within a given process. Choreography is the basis for the Service Oriented Architecture (SOA) paradigm and helps to keep the enterprise loosely coupled and agile. The choreography of a process focuses on the responsibilities and interactions that ultimately provide value without necessarily requiring any coordinating authority.

In business process modeling, choreography and orchestration are effectively two sides of the same coin. BPDM joins orchestration and choreography into a unified and coherent model.

## 1.1 Business Process Modeling Notation (BPMN)

BPMN has gained recognition as a flexible and business-friendly notation for process orchestration. BPDM provides an explicit metamodel and serialization mechanism for BPMN concepts. By integrating BPMN and BPDM both the underlying model and notation for process orchestration is covered by an integrated set of standards. The notation for choreography, BPMN diagram interchange and the normative relationship to runtime technologies such as BPEL is planned to be part of subsequent standards.

## 1.2 Target Audience and Use of BPDM

At its core, BPDM provides interoperability across tools, so that different tools can depict or utilize a process definition in different ways yet work together for the ultimate benefit of the enterprise. For example, If Vendor A and Vendor B both support BPDM as their process exchange mechanism, then, a BPMN drawing created using Vendor A's modeling tool could then be opened and executed using Vendor B's business process management system. Therefore, BPDM is a technology specification for vendors to use to define how they serialize or exchange their process depictions, allowing for industry interoperability. For most business analysts and process users, this is all they really need to know about BPDM. What BPDM support means is that your process assets are not locked into a particular tool or notation; they are assets that can work across a wide range of tools and solutions.

## 1.3 Other Common Business Benefits of BPDM

### 1.3.1 Carefully defined semantics

When diagrams are used to aid human to human communications a certain amount of "fuzziness" in what those notations mean can be acceptable, since explanations often clear up any misunderstandings. When processes are specifications for what people, organizations or I.T. systems should do, those specifications must be clear and precise. Particular attention has been paid in BPDM to make sure that the semantics behind the notations and models are well defined, consistent and sufficient to represent most normal forms of business processes. BPDM is sufficiently precise to model behavioral changes/events (starting, ending, aborting, etc) of processes that allows them to be ordered in time, and have their effects on each other precisely modeled. Formal methods , based on logic, are utilized to verify this precision. The precise semantics of BPDM makes sure that processes will be accurately communicated to man and machine.

### 1.3.2 Saying just enough, but not too much:

Specifying a business process can be a double-edged sword. Say too little and the process may be unpredictable, inconsistent, wasteful and not fit into the rest of the business (or the business of partners). Say too much and the process can be a strangle-hold, preventing creativity, agility and optimization. BPDM can't enforce this artful balance, but it can enable it; the basis of which is separation of concerns - separating the intended outcome of a process from how that outcome is achieved. Where appropriate; substantial detail can be specified for how to achieve a goal, in other cases only the "contract" is specified - the contract says what is to be accomplished without saying how. Many of the established methods do not provide well for this separation of concerns and therefore over specify or under specify a process. BPDM provides for separation of concerns, well defined contracts and multiple options for implementing a process that correspond to its contracts.

### 1.3.3 Improved Integration and Collaboration

The successful modern enterprise is defined by two basic capabilities; the ability to be agile and the ability to collaborate. Both capabilities are served by "loosely coupling" the business and the technologies that serve it. This means that tightly coupled and monolithic processes are barriers to success. A business process design better serves the enterprise by making it easy to collaborate with other organizations, regardless of their processes. It should be easy to outsource, insource or change the way a part of the organization works without undue impact on the rest of the organization or business partners. The integration of orchestration and collaboration as well as the separation of process contract from its realization serve this goal of loose coupling.

### 1.3.4 Improved Agility

Agility is required to respond to external drivers, internal needs and the constant impact of legislation and technology change. In today's' world - agility is survival. The combination of well defined business processes that provide for separation of concerns with Model Driven Architecture (MDA)® provide the exciting possibility of being able to design, redesign and deploy new processes quickly and with minimal overhead - the enterprise is not locked in to legacy technologies and processes. BPDM provides the business focused model that can be part of the specification of the process for people, in terms of process "play books" and instructions, and for technologies, in terms of web services, workflows and process execution engines. In addition BPDM is technology independent - any number of technical approaches may be used to help realize or support a business process. The BPDM model is a model of the business, not the technology - MDA helps join these two viewpoints.

### 1.3.5 Business Processes supported by Service Oriented Architectures (SOA)

SOA has become recognized as the leading architectural approach to business and technical agility and integration. SOA structures the enterprise and supporting technologies based on services that are provided or consumed by collaborating entities. This service oriented approach applies to both the business - in terms of how one business or business unit serves another, and to the technologies - in terms of how application components work together by providing and using software services. BPDM describes the business side of SOA in terms of choreography (above) that can then be mapped to the software components that assist those business processes. This process centric SOA approach provides for agility, loose coupling and a better tie between business and technology. SOA helps support both the agility and collaboration goals of BPDM.

### 1.3.6 Better Return on I.T. Investment

The net result of separation of concerns, support for collaboration and enhanced agility is that I.T. investments have better return. This return is realized by directly supporting business needs as identified in the business processes and by supporting reuse of services, components and supporting infrastructure across the enterprise and across marketplaces. Since investments are more reusable, their return is not limited to a single project. Since investments are directly tied to business needs, their business benefit can be measured. Since investments support agility and collaboration, they can have bottom-line impact.

## 1.4 Process Concepts supported by BPDM

BPDM integrates multiple process approaches and notations, which are summarized as follows. BPDM provides integrated and consistent support for the semantics of:

- All BPMN notation concepts
- Processes, activities, tasks and sub-processes
- Workflow
- Sophisticated control of alternatives and parallel processes
- Conditional execution paths
- Signals and events
- Time-based events and conditions
- Events based on change in data or external conditions
- Integration with rules and rules engines through event-based semantics
- Process groups and swim-lanes

- Transactions, rollback and compensation
- Process data and data flow
- Artifacts and artifact production and dependencies
- A combination of human and automated process participants
- Service Oriented Architectures and business services
- Resource and entity selection
- Roles, responsibilities & collaborations
- Bi-directional and composite interactions between entities
- Automated execution with MDA and process execution engines such as BPEL (See non-normative mapping to BPEL)
- Interaction protocols, services and design by contract
- Composite processes
- UML activity, collaboration and interaction diagram concepts
- Process specialization, derivation and refinement

In summary, BPDM standardizes the underlying semantics, model and exchange mechanisms to improve the efficiency, agility and collaboration of public and private enterprises through the precise and integrated definition of business processes.

# 2 Conformance

The following levels of compliance are defined for BPDM in relation software. For the following compliance points the interpretation of the phrase "to process a model" will depend on the functionality of the software as follows;

- If the software reads process models, to "process the model" will include reading a BPDM model compliant with the MOF-2 XMI for BPDM included as part of this specification.
- If the software writes process models, to "process the model" will include writing a BPDM model compliant with the MOF-2 XMI for BPDM included as part of this specification.
- If the software executes or otherwise interprets process models, to "process the model" will include executing or interpreting the model in accordance with the semantics as defined in this document.

## 2.1 BPDM Full Compliance

An implementation is fully compliant if it can process a model that utilizes all BPDM metamodel concrete concepts, not necessarily including those defined in the “BPMN Extensions” package.

## 2.2 BPDM Collaboration Protocol Compliance

An implementation is BPDM protocol compliant if it can process a collaboration protocol model that utilizes all concrete concepts for representation of a collaboration protocol as specified in the “Interaction Protocol Process Model” package and all included packages.

## 2.3 BPDM Orchestration Process Compliance

An implementation is BPDM protocol compliant if it can process an orchestration model that utilizes all concrete concepts for representation of a orchestration process as specified in the “Activity Model” package and all included packages.

## 2.4 BPDM - BPMN Compliance

An implementation is BPMN compliant if it can process a model that utilizes all concrete concepts for representation of a process as specified in Section 4.4, 4.5, 4.6, 4.7 and 4.8. Each of these sections provides a detailed mapping of the BPMN constructs on to the BPDM metamodel.

# 3 Normative References

| | |
|---|---|
| [BPEL11] | ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf |
| [BPEL20] | http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf |
| [BPMN] | http://www.omg.org/docs/dtc/06-02-01.pdf |
| [BPM-06-02] | http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-02.pdf |
| [RFC2119] | http://www.ietf.org/rfc/rfc2119.txt |

# 4 Terms and Definitions

### Activity

An **Activity**  is a kind of **Processing Step** that activates a **Processing Behavior** (it operates over time) in the context of a **Process**. It can:

- be ordered in time by **Processing Succession**
- operate under the responsibility of a **Performer Role**
- activate a sub-processe or be a simple task that start and stop

An **Activity**  is also an **Interactive Part** that receives its inputs and outputs through **Interactions** coming from other **Interactive Part**s in the **Process** (**Activity**, **Interaction Role**, **Performer Role**, **Holder**).

### Actor

An **Actor** is an entity that is responsible for the execution of duties specified by a **Performer Role**

Further sub-type of **Actor** will be defines in specifications such as the the Organizational Structure Metamodel (OSM) to add specific requirements such as  and can

as having certain skills or budget.

### Performer Role

A **Performer Role** is a **Part Group** that takes responsibility of performing activities in the process. Being an **Interactive Part**, a **Performer Role** also has responsibilities to fulfill **Interactions** that it is involved with other **Performer Roles** or with **Interaction Roles** at the boundary of the **Process**. A **Performer Role** is a **Typed Part** for specifying **Actor** that can play the role at process enactment.

A **Performer Role** can be decomposed into sub **Performer Role** to delegate responsibility for a subset of its activities or interactions. A **Performer Role** may have a realization as defined by a **Role Realization** that further specifies how the **Performer Role** will meet its responsibilities.

### Process

A  **Process** is a kind of **Processing Behavior** that describes specific **Activity**(ies) to be performed, **Interaction**s to be undertaken during its execution under the authority of a **Processor Role** (or **delegated performer role** s).

The process owns the set of activities to be performed as well as the **Condition**s on when such activities will be performed and by which performer role.  The process also owns the set of **Interactive Part**s that define the flow of information and other resources between activities,**Performer Role** and **Interaction Role**s.

A specific **Interaction Role** defines the set of **Interaction**s the process is responsible of: its is the **Process Interaction Boundary**. The set of **Interaction**s attached to the **Process Interaction Boundary** defines the inputs and outputs of the process

A  Process may utilize sub-processes with a **Sub-Process Activity** as well as be used in the context of other processes in the same way.

### Condition

A **Condition** determines if the semantics of a model element applies or not during the enactment of a process.

In the user's model it is a boolean expression. During the (enactment, execution) occurrence of a process it is evaluated to determine if the semantics of the model element applies at the time of the evaluation.

### Succession

A **Succession** is a **Directed Part Connection** that organizes **Course Parts** in series in the context of a **Course**. A **Succession** indicates that  that one **Course Part** "follows" another in time, and possibly establishes constraints on such followings.

### Behavioral Change

A **Behavioral Change** is a kind of **Change** that occurs as part of the lifecycle of a **Behavioral Happening**, such as **Start**, **Finish** or **Abort**.

BPDM provides a predefined library of **Behavioral Changes**.

### Behavioral Change Part

A **Behavioral Change Part** identifies **Behavioral Change** (such as **Start** or **End**) for an individual **Behavioral Happening**.  A **Behavioral Change Part** is also a **Course Part**, enabling it to be connected by **Successions**.

## Behavioral Happening

A **Behavioral Happening** is a kind of **Happening Over Time**that produces **Behavioral Changes** which are behavior lifecycle events, such as **Start** and **End**. A **Behavioral Happening** is also a **Course**, enabling its lifecycle to be ordered in time by **Successions** .

A user (M1) library - **Happening & Change Library** - captures commonly needed aspects of **Behavioral Happenings** as instances such as the finish being after the start.

## Change

A **Change** is a **Type** for dynamic entities occurring at a point in time.

## Change Condition

A **Change Condition** is a **Condition** for specifying that a **Change** must occur in the context of a particular **Happening Over Time** for the condition to hold.

For instance, a condition can be on the eruption (instance of **Change**) of a particular volcano (instance of **Happening Over Time**).

## Behavioral Step Group

A **Behavioral Step Group** is a kind of **Part Group** that is also a **Behavioral Step** typed by the **Universal Behavioral Happening** in user models (M1). This gives a group of **Behavioral Steps** as a whole the capacity to produce start and end changes playing the standard behavioral change parts, such as **startPart** and **endPart**.

For example, most process languages have a way of modeling sub-processes without defining a separate process. This is a **Behavioral Step Group**.

## Change Condition Step

A **Change Condition Step** is a kind of **Typed Course Part** that monitors the occurrence of a **Change Condition** and that has an effect on the course of a **Processing Behavior**. For instance, a **Change Condition Step** can be used to react to the **Abort** of a specific **Behavioral Happening**.

## Processing Succession

A **Processing Succession** is a kind of **Succession** that can order the **Behavioral Change Parts** of its **Behavioral Step** such as their start and end parts.

Processing Succession allows any combination of Behavioral Change Part to be connected.

End -> Start

Start -> Start

Start -> Abort

etc.

A **Processing Succession** doesn't need to have **Behavioral Steps** on its ends, it can have untyped course parts also, such as **Course Control Part**, but it must have something on each end, as all **Successions** do.

For convenience, a **Processing Succession** that do not specify **predecessor behavioral change part** or **successor behavioral change part** will have the same effect as a **Processing Succession** where these are respectively the end part and start part.

### Interaction

An **Interaction** is a **Behavioral Step** that is also a **Part Connection** , enabling **Interaction** to have start and end changes, and be ordered in time.

An **Interaction** can be either a simple **Simple Interaction** or a set of combined **Simple Interaction**s : a **Compound Interaction**. Ultimately, an **Interaction** is realized by the exchange of **Simple Interaction**s between its **Interactive Part**s.

### Interaction Role

An **Interaction Role** is an **Interactive Part** where the individuals playing the part are in the environment context where the **Processing Behavior** is used. For example, the customer is an **Interaction Role** in a behavior for delivering a product.

### Simple Interaction

An **Simple Interaction** is a kind of **Interaction** in which something is "transferred" from individuals playing one interactive part to individuals playing another interactive part. For example, a document, phone number, or package may be transferred from one department to another in a company. The transferred items must conform to a **Type** specified by the **Simple Interaction**. A **Simple Interaction** can have an **Expression** to change the item that arrives at the target based on the item flowing from the source. For example, a transformation may retrieve the zip code from an address flowing from the source to deliver the zip code to the target.

**Simple Interactions** in user (M1) models are always typed by the **Universal Behavioral Happening** (see user library **Happening & Change Library**). This gives them the standard **Behavioral Change Parts**, such as for start and end, so the **Simple Interactions** can be ordered within an **Interaction Protocol**. This is different from the type of thing transferred.

**Simple Interactions** can refer to **Simple Interactions** inside the **Interactive Parts** being connected. This means the transferred thing is passed along through chains of **Simple Interactions** from inside to outside the parts, or the other way (see **Simple Interaction Binding**)

### Interaction Protocol

An **Interaction Protocol** is a kind of **Processing Behavior** where **Behavioral Steps** are **Interactions** that occur between **Interaction Roles**.

The set of **Interaction**s defines the purpose of the **Interaction Protocol**.

# 5 Additional Information

## 5.1 Acknowledgements

The following companies submitted this specification:

- Adaptive
- Axway Software
- Borland Software
- Model Driven Solutions
- EDS
- Lombardi Software
- MEGA International
- Unisys

The following companies and organizations support this specification:

- BPM Focus
- U.S. National Institute of Standards and Technology (NIST)

# 6 Metamodel Specification

This chapter presents the normative specification for business process definition metamodel. It begins with an overview of the BPDM metamodel structure followed by a description of each sub-package.

## 6.1 Overview

The BPDM package contains the models for orchestration (including BPMN) and choreography, and their performance, enactment, and execution. It has eight subpackages grouped into three categories:

- Common Abstractions for the framework that ties the other models to performance, enactment, and execution (Composition Model and Course Model).
- Common Behavior for the aspects of dynamics in common between orchestrations and choreography (Happening & Change, Processing Behavior, and Simple Interaction).
- Activity Model (including BPMN Extensions) for orchestration and Interaction Protocol Model for choreography.

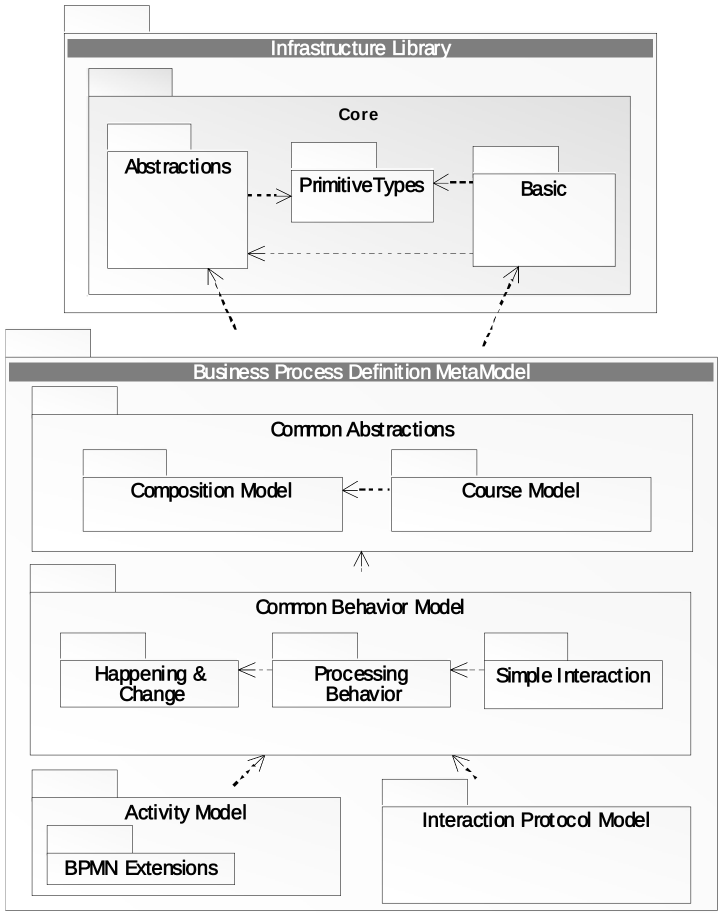


**Figure 6.1 - Dependencies of BPDM Packages**

| **Package** | **Comment** |
|---|---|
| Business Process Definition MetaModel | The BPDM package contains the models for orchestration (including BPMN) and choreography, and their performance, enactment, and execution. It has eight subpackages grouped into three categories:<br>Common Abstractions for the framework that ties the other models to performance, enactment, and execution (Composition Model and Course Model).<br>Common Behavior for the aspects of dynamics in common between orchestrations and choreography (Happening & Change, Processing Behavior, and Simple Interaction).<br>Activity Model (including BPMN Extensions) for orchestration and Interaction Protocol Model for choreography. |
| Common Abstractions | The Common Abstractions groups the packages that provides abstractions used in other packages. |
| Composition Model | The Composition Model is a framework for relating metamodels to the real world entities they ultimately represent. It facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and execution across business process management suites. The Composition Model enables users and vendors to build libraries of orchestrations and choreographies, including specialization of some orchestrations or choreographies from others. It also enables users and vendors to define their own frameworks for recording data about ongoing orchestrations and choreographies, for example, how long they have been going, who is involved in them, and what resources they are using. |
| Course Model | The Course Model extends the Composition Model to connect parts in time (Succession). For example, a succession connects one step in a process to another to indicate that the second step happens after the first. The same applies to messages in choreography. |
| Common Behavior Model | The Common Behavior Model includes elements shared by all process oriented behavior models |
| Happening & Change | The Happening and Change Model introduces dynamics, in particular, time ordering of lifecycle events, such as starting and ending a process. This facilitates the integration of rule and monitoring systems with models of dynamics, such as orchestration and choreography. The model enables users and vendors to define their own libraries of processes, with their own categorizations and attributes, such as how long a process has been running, and the resources it is using. They can also define their own life cycle events, for example, to define finish statuses and taxonomy of errors. |

| | |
|---|---|
| Processing Behavior | The Processing Behavior Model enables behavioral happenings to be ordered in time as parts of other behavioral happenings (see the Happening and Change model). Vendors and users can define their own execution patterns with connections between these behavioral parts. The model predefines a specific connection for races, where behavioral happenings start at the same time and abort each other when the first finishes.  It also defines a change condition for detecting lifecycle events in behavioral happenings. The Processing Behavior Model is the most specialized model in BPDM that still covers all of processes and interactions (orchestration and choreography, see the Activity and Interaction Protocol Models). |
| Simple Interaction | The Simple Interaction Model enables interactions to be treated like any other step in a processing behavior, ordered in time, with start and end events.  The model is the basis for flows between process steps and between participants in a choreography (see the Activity Model and the Interaction Protocol Model). The Simple Interaction Model is the most specialized model in BPDM that still has elements in common between processes and choreographies. |
| Activity Model | The Activity Model is for capturing orchestrations in way that facilitates modification as boundaries of process of business change, for example, due to insourcing, outsourcing, mergers, and acquisitions. It uses interactions to represent inputs and outputs, enabling choreographies to be specified between the process and its environment, as well as between the performers responsible for steps in the process. The Activity Model is the basis for the BPMN model in BPDM (see the BPMN Extensions). |
| BPMN Extensions | The BPMN Extension provides additions to the Activity Model for BPMN. These provide BPMN names for special usages of BPDM concepts and additional functionality specific to BPMN |
| Interaction Protocol Model | The Interaction Protocol Model is for capturing choreographies.  It enables interactions to be grouped together into larger, reusable interactions. For example, an interaction that exchanges goods between companies might be used with other interactions within a larger protocol representing a partnership of the companies.  This protocol might be adopted by a standards body and reused between many pairs of companies. The interactions in a protocol may be simple interactions that have no sub-interactions, or may be other protocols. |
| Infrastructure Library | InfrastructureLibrary package defines a reusable metalanguage kernel and a metamodel extension mechanism for UML. The metalanguage kernel can be used to specify a variety of metamodels, including UML, MOF, and CWM. In addition, the library defines a profiling extension mechanism that can be used to customize UML for different platforms and domains without supporting a complete metamodeling capability. |
| Core | The Core package is the central reusable part of the InfrastructureLibrary. |
| PrimitiveTypes | The PrimitiveTypes package of InfrastructureLibrary::Core contains a number of predefined types used when defining the abstract syntax of metamodels. |
| Abstractions | The Abstractions package of InfrastructureLibrary::Core is divided into a number of finer-grained packages to facilitate flexible reuse when creating metamodels. |

| | |
|---|---|
| Basic | The Basic package of InfrastructureLibrary::Core provides a minimal class-based modeling language on top of which more complex languages can be built. It is intended for reuse by the Essential layer of the Meta-Object Facility (MOF). The metaclasses in Basic are specified using four diagrams: Types, Classes, DataTypes, and Packages. Basic can be viewed as an instance of itself. More complex versions of the Basic constructs are defined in Constructs, which is intended for reuse by the Complete layer of MOF as well as the UML Superstructure. |

## 6.2 Composition Model

### 6.2.1 Introduction

The Composition Model is a framework for relating metamodels to the real world entities they ultimately represent, in particular those with interconnected elements in the same organized whole. This facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and execution across business process management suites. The Composition Model enables users and vendors to build libraries of orchestrations and choreographies, including specialization of some orchestrations or choreographies from others. It also enables users and vendors to define their own frameworks for recording data about ongoing orchestrations and choreographies, for example, how long they have been going, who is involved in them, and what resources they are using.

The Composition Model provides general capabilities for representing:

1. The interconnection of elements due to their relation to the same other element. For example, the steps in a process are interconnected because they are all parts of the same process. Interconnections can differ by the element they have in common. For example, two processes might have the same steps, but in a different order.

2. Interconnections that are composed of other interconnections. For example, the many fine-grained communications between businesses to set up a partnership may be aggregated into a single joint choreography when viewed at a high level.

3. Interconnections between interconnections. For example, when one communication happens before another during a choreography, it is a connection in time between two other connections.

4. User and vendor-defined characteristics of elements, such as cost, person responsible for them, and resources being consumed.

The Composition Model can be applied in many domains, including structural ones, but in BPDM it is applied to modeling of dynamics, specifically to orchestration and choreography. In this domain the elements are steps in orchestrations, or interactions in choreographies, and the interconnections are relationships in time or transfers of information or physical objects between elements. The elements of the Composition Model are specialized in the other BPDM packages for application to these areas.

The first subsection below is the basis for applying BPDM to business process execution and rules, and to understanding the specification in general. The remaining subsections cover the major elements of the Composition Model.

### 6.2.1.1 Individuals, Models, and Modeling Languages

An individual is any uniquely identifiable thing. For example, it can be an organization, a piece of hardware, or software component, or something more ephemeral like an information object, process, interaction, or event. The only requirement is that it is distinguished from other individuals. Individual processes and interactions occur at particular times, and are variously called performances, enactments, or executions.

A model describes what we would like from individuals (the *model semantics*). For example, a model of a business specifies what is desired from an actual real world business. Some businesses will satisfy these desires, some will not. Individuals that satisfy the model are said to conform to the model. The rules for conformance are the semantics of the model.[1]

A modeling language consists of shorthands for expressing the semantics of a model. Shorthands used in a model can be "expanded" to give the semantics. For example, a common semantic pattern is to say that all individuals of one kind are also of another kind. A shorthand for this is sometimes called "generalization". Generalization might be used in a model to say that businesses are a generalization of small businesses. This is a shorthand for saying any individual that is a small business is also a business.[2]

Individuals exist at the M0 level in OMG's Model Driven Architecture, while models exist at the M1 level, and modeling languages at the M2 level. The term "individual" in this specification refers only to elements that are not in models or modeling languages, even though the contents of models and modeling languages are uniquely identifiable like any individual. Similarly, the term "model" in this specification refers only to elements that are not individuals or modeling languages, even though a model language may be expressed as a model (metamodel, see below). More examples and explanation are available in Sections 7.9 through 7.12 of the UML Infrastructure, http://doc.omg.org/formal/07-02-06.

A modeling language has two parts:

- The *language syntax* gives the names of the modeling shorthands and how they can be combined. For example, generalization applies between exactly two kinds of things. Syntax alone cannot determine model semantics, because it refers only to model elements, not individuals.[3]
- The *language semantics* specifies how shorthands are expanded into model semantics. For example, generalization in a model expands to individuals of one kind of thing in the model also being individuals of the other. Language semantics builds on syntax, but must refer to individuals to give a syntax its M0 meaning when the syntax is used in a model.

Some syntaxes are better for specifying language semantics than others. In particular, a syntax that identifies model elements categorizing individuals provides a better basis for specifying model semantics. This enables the language semantics to refer to individuals via the model elements that categorize them. Following the UML Infrastructure terminology, BPDM calls these syntactical elements "Types."

---

1. The phrase "instance of" is sometimes used to mean the conformance of an individual to a particular model element (which is often called a "class"), but this terminology usually refers to classes as factories for creating instances, rather than classes as categories. For example, if an individual Fido is a Dog, then Fido is also a Mammal, so conforms to both Dog and Mammal, even though normally Fido would not be called an instance of Mammal, because it was not "created" from Mammal.
2. The difference between shorthands and templates is that the expansion of templates are captured in a machine-understandable way, as part of the modeling language. The expansion of shorthands are specified less formally. Shorthands are more susceptible to misinterpretation than templates, leading to communication failures between users and lack of interoperability between tools.
3. A metamodel specifies syntax by omitting some aspects of the graphical or textual appearance of the language, such as geometric shapes or punctuation. For example, a metamodel might have an element for kinds of things and another for generalization, but no mention of how generalization appears in a graphical or textual language. This is sometimes called "abstract syntax", as distinguished from "concrete syntax", which includes the detailed graphical or textual appearances.

#### 6.2.1.2 Types

Types group individuals according to some commonality among them, which might be characteristics they can have or constraints they obey. Types can cover any kind of entity, physical or computational, static or dynamic. For example, the type Person groups individual people, like Mary and John. The type declares commonalities among people, for example, they can have names and gender, or obey constraints, such as being genetically related to exactly two other people.

Types can group individual occurrences of dynamic entities (M0), such as processes and interactions. For example, the type Order Process groups individual performances, enactments, or executions of the ordering, where each occurrence happens between particular start and end times. The type declares commonalities among the occurrences, for example, that they involve a product or service, or obey constraints, such as having certain steps taken in a certain order.

#### 6.2.1.3 Composites

Composites are Types specifying the interconnections of individuals that are all related to the same other individual (M0). For example, a company type specifies the interconnections of departments within each individual company of that type (assuming it is modeled in a value chain manner, rather than just an organization chart). Likewise, an orchestration type specifies the sequence of steps in each individual occurrence of that orchestration.

The things interconnected by a composite can have any kind of relation to the composite. They are not necessarily "contained", "owned", or "part of" the composite. For example, choreographies are composites with the communicating businesses entities as "parts", but the businesses entities are not contained by the choreography in any sense.

#### 6.2.1.4 Parts

To clarify the meaning of "Part" in BPDM, it is important to distinguish two senses in ordinary English:

- Part as an individual, for example the Acme Furniture Company with a unique tax identification number.
- Part as a role, as in "part in a play."

These are mutually defining. Parts in the first sense (individuals) play parts in the second sense ("roles"). For example, a person Mary (individual) may play the president (role) in the Acme Furniture Company. Roles map an individual whole into another individual playing that role in the whole. For example, the president role maps Acme Furniture Company to Mary. (The term "role" is used informally in this section. It has a more specialized meaning in other packages of BPDM.)

Typed Parts in BPDM have the second meaning above. Individuals playing a typed part must be of a certain kind (Type), and play the part in the context of another type of thing (whole). For example, an individual playing the president part must be a person, and must play the president within an individual company.[4] Individuals playing parts can have any relation to the whole. They are not necessarily "contained," "owned," or "part of" the whole. For example, a person might be modeled as a composite of anatomically contained parts, but still have other typed parts for relations to other people, such as spouses. The typed part spouseOf will have individuals playing that role for other individuals, but the people are not contained within each other. Typed Parts are MultiplicityElements for restricting the number of individuals that play the part. For example, a company might allow no more than five vice-presidents, but require a president, and a choreography might have an interaction that is optional.

4. Typed parts are equivalent to what are sometimes called "properties" or "attributes." In this terminology, an individual playing a part is called the "value" of the property or attribute.

Parts in BPDM are a generalization of Typed Parts to include elements in a composite that do not correspond to individuals (identifiable M0 entities, see Individuals, Models and Modeling Languages). For example, process models often have an indicator that some steps happen at the same time. This part of a process model does not correspond to anything identifiable in the M0 occurrences of the process. It just models the constraint that there are suboccurrences happening at the same time. Because of this, these parts do not have a type restriction like Typed Parts do.

Part Groups are Parts that collect together other Parts. Part groups can share parts. The meaning of part groups is given in the specializations of the Composition Model, for example, in Processing Behavior.

#### 6.2.1.5 Part Connections

Connections between typed parts in the composition model specify links between M0 entities playing the typed parts. For example, the reporting connection between the president of a company and the CEO means the person playing the president in a particular company will report to the person playing the CEO in the same company. Likewise, the temporal connection between one step and another in a process means that in each occurrence of that process, there is an occurrence of one step that happens after the occurrence of another.

Connections involving untyped parts do not have a predefined meaning in the Composition Model. They are given specialized interpretations in other packages of BPDM, depending on the parts being connected. For example, parts of a process model indicating that some steps happen at the same time are untyped. Connections to and from these parts require special interpretation to reflect this intention.

Part Connections can be treated as first-class parts in themselves, by defining classes that are subtypes of both Part Connection and Typed Part, as done in other BPDM packages. This provides connections that have parts, and connections to connections. For example, choreographies are connections between business entities that are composed of many communications between the businesses. These communications are connections also, and occur in a certain order, which are temporal connections between the communications. Choreographies are the type of their M0 performances, enactments, or executions, which are also M0 links between the businesses. Typed connections require the modeler to specify which parts of the type correspond to which parts on the ends of the connection, see the Part Binding subsection below.

Directed Part Connections are Part Connections between two parts that facilitate traversal from one to the other in user (M1) models. Their source and target associations specify the top-level parts (not part paths) that are connected, as typically shown by the arrows in process diagrams. For example, when one step is after another in a process, the arrow between them is modeled as a directed connection, with the earlier step at the source end, and the later step at the target end. Connections in general can connect any number of parts. For example, a business interaction can involve multiple companies.

Conditions may be applied to connections to limit when they apply. For example, one step in a process may happen after another only when certain conditions are true as the process is executing. Opaque Conditions are Conditions that enables the condition to be expressed textually in multiple languages. Irreflexive Conditions are for restricting connections to apply at M0 only between distinct M0 individuals playing the part (or playing the last part in the path). It applies only to connections between typed parts, or paths with at least one typed part. Compound Conditions provide for combining other conditions with Boolean operators, such as “and” and “or.”

#### 6.2.1.6 Part Paths

Some connections are between parts of parts. For example, the temporal connections between steps in a process typically indicate that the start of one step is after the end of another, but they might also indicate that the start of one step is after the start of another, or the end of one step is after the end of another, and so on. To distinguish these cases, the parts on each end of the connection must specify which event (start, end) it is referring to “inside” the step on that end.[5] In BPDM individual events at M0 can be identified by parts, and the combination of the step and the event part is a Part Path.

Part Paths are Parts that enable connections to refer to parts of parts, for example to connect the end and start events in two steps of a process. For generality, it enables connections to refer to parts of parts to any depth. For example, a part path might refer to the time at which the start event in a step occurs, where the time of an event is modeled as a part of the event. This defines a path through three parts.[6] Part Paths can have a short cut to the last element in the path (final target), for convenience. Part Paths and Parts are generalized to Abstract Parts, which are the ends of connections. This enables connections not requiring part paths to refer directly to parts, rather than to part paths with only one element.

#### 6.2.1.7 Generalization and Derivation

Generalization is a relationship between Types indicating that M0 individuals of one type are also individuals of another type. For example, business is a generalization of small business because individual small businesses are also individual businesses. Specialization is the opposite of generalization, for example, small business is a specialization of business. Parts and constraints specified on the general type apply to all individuals conforming to specializations of that type, because those individuals also conform to the more general type. For example, businesses in general attempt to make a profit, so small businesses do also.

Derivation is a relationship between Composite Types that replaces some parts with others. There is no restriction on the number or kinds of parts that can be replaced by a derived composite. Derivation is useful for exploring alternative configurations for a composite. There are no parts or constraints specified on a composite type that are guaranteed to apply to individuals of derived types.

#### 6.2.1.8 Selection

A selector specifies the individuals playing a Typed Part. This might be determined by a rule for each M0 whole that contains the part. A special kind of rule is that the individual must be drawn from a set of predetermined individuals.

### 6.2.2 Metamodel Specification

The Composition Model is a framework for relating metamodels to the real world entities they ultimately represent. It facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and execution across business process management suites. The Composition Model enables users and vendors to build libraries of orchestrations and choreographies, including specialization of some orchestrations or choreographies from others. It also enables users and vendors to define their own frameworks for recording data about ongoing orchestrations and choreographies, for example, how long they have been going, who is involved in them, and what resources they are using.

5. The step must be specified as a part, rather than just the type of thing done at the step, because a process might have more than one step that does the same thing.
6. A path can contain at most one untyped part, which must be at the end of the path, otherwise it would not be possible to navigate through to the end of the path.

### 6.2.2.1 Composition

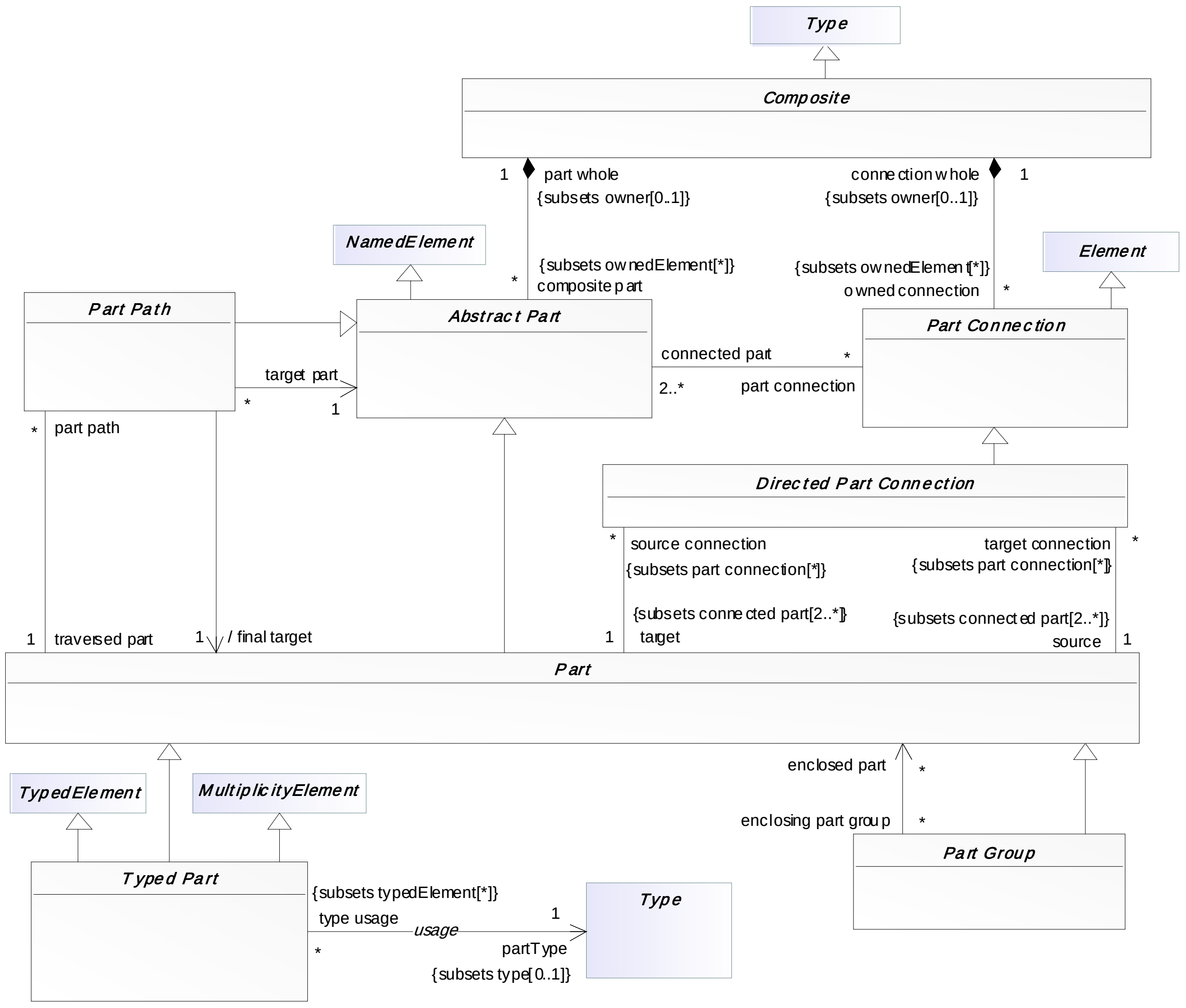


**Figure 6.2 - Composition**

### 6.2.2.2 Part Connection & Condition

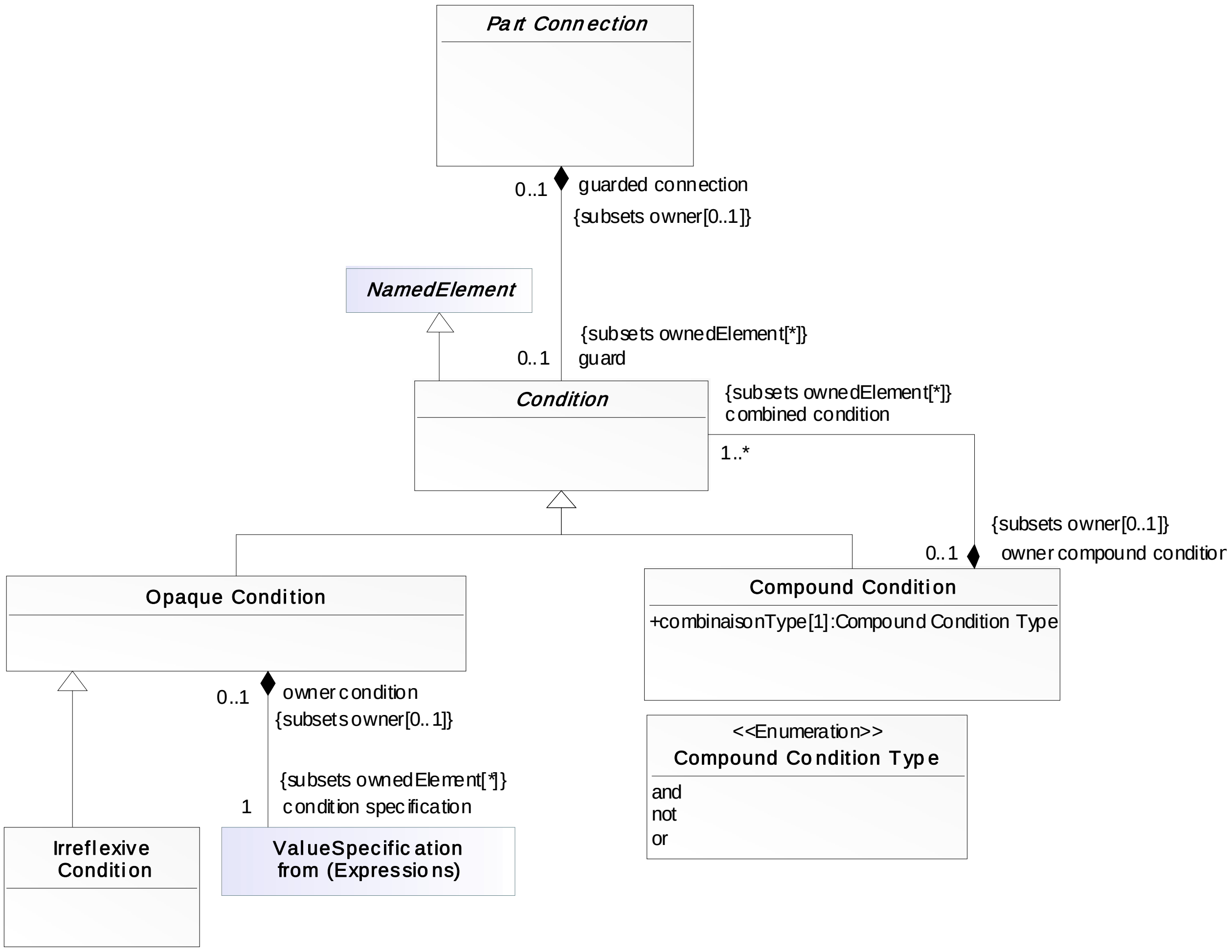


**Figure 6.3 - Part Connection & Condition**

### 6.2.2.3 Generalization & Derivation

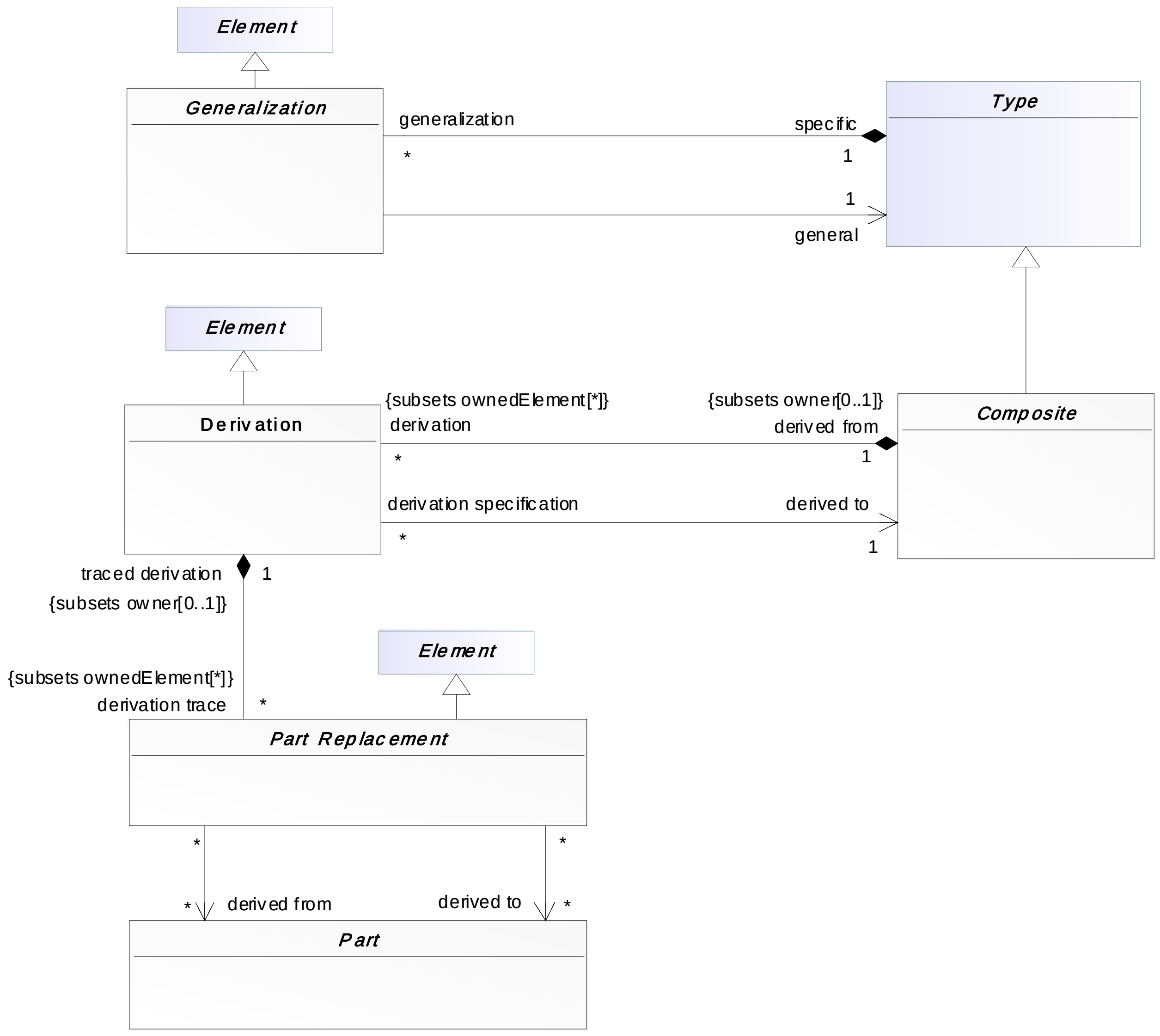


**Figure 6.4 - Generalization & Derivation**

### 6.2.2.4 Selection

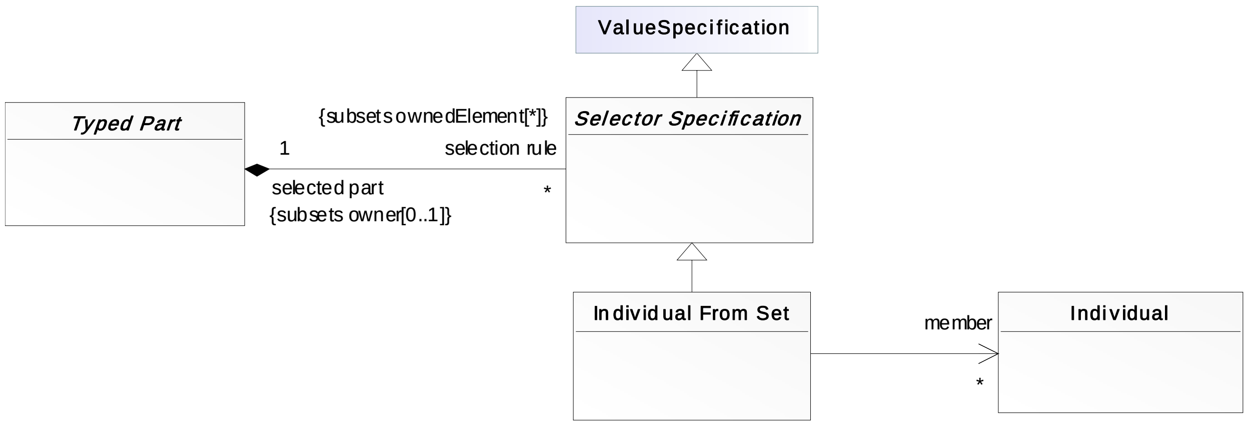


**Figure 6.5 - Selection**

### 6.2.2.5 Abstract Part

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** “NamedElement”

#### Description

**Abstract Part** is the subject of relations between parts through **Part Connection**. **Abstract Part** is a capability that all kinds of **Part**s share.

Individuals playing parts can have any relation to the whole, they are not necessarily “contained,” “owned,” or “part of” the whole.

#### Associations

| | |
|---|---|
| part connection : Part Connection [*] | Connection connecting the Part to one or more other Parts. |

### 6.2.2.6 Composite

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** “Type”

#### Description

A **Composite** is a **Type** which has an internal structure. It specifies the connections of individuals that are all related to the same other individual (M0). For example, a company type specifies the connections of departments within each individual company of that type (assuming it is modeled in a value chain manner, rather than just an organization chart). Likewise, an orchestration type specifies the sequence of steps in each individual occurrence of that orchestration.

### Associations

| | |
|---|---|
| composite part : Abstract Part [*] | Part owned by the Composite<br>Subsets *ownedElement* |

| | |
|---|---|
| derivation : Derivation [*] | Derivation that the Composite is a source of<br>Subsets *ownedElement* |

| | |
|---|---|
| owned connection : Part Connection [*] | Part Connection owned by the Composite<br>Subsets *ownedElement* |

### 6.2.2.7 Compound Condition

**Namespace:** Composition Model
**isAbstract:**
**Generalization:** “Condition”

#### Description

A **Compound Condition** is a kind of **Condition** that is the combination of other **Conditions**. There are three kinds of **Compound Condition**:

- **or**: the **Compound Condition** is the result of of one the **combined condition**
- **and**: the **Compound Condition** is the result of all the **combined condition**
- **not**: the **Compound Condition** is result of the negation of all the **combined condition**

#### Attributes

| | |
|---|---|
| combinaisonType: Compound Condition Type [1] | |

### 6.2.2.8 Compound Condition Type

**Namespace:** Composition Model
**isAbstract:**

#### Description

Enumeration specifying the different types of **Compound Condition**

| | |
|---|---|
| and: | |

| | |
|---|---|
| not: | |

| | |
|---|---|
| or: | |

### 6.2.2.9 Condition

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** “NamedElement”

#### Description

A **Condition** determines if the semantics of a model element applies or not during the enactment of a process.

In the user's model it is a boolean expression. During the (enactment, execution) occurrence of a process it is evaluated to determine if the semantics of the model element applies at the time of the evaluation.

### 6.2.2.10 Derivation

**Namespace:** Composition Model
**isAbstract:**
**Generalization:** “Element”

#### Description

The **Part**s of the **derived to Composite** are the same as the on **derived from Composite**, except for replaced or removed **Part**s, as specified by **derivation trace**, or added parts.

#### Associations

| | |
|---|---|
| derivation trace : Part Replacement [*] | Part Replacement owned by the Derivation<br>Subsets *ownedElement* |

| | |
|---|---|
| derived to : Composite [1] | Derived Composite |

### 6.2.2.11 Directed Part Connection

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** “Part Connection”

#### Description

A **Directed Part Connection** is a kind of **Part Connection** for only two parts, when it is convenient to have standard names referring to the parts on each end (source and target).

**Directed Part Connections** are designed to facilitate traversal of **Part Connections** . Their source and target associations specify the top-level parts (not **Part Paths**) that are connected, as typically shown by the arrows in process diagrams. For example, when one step is after another in a process, the arrow between them is modeled as a directed connection, with the earlier step at the source part, and the later step at the target part.

**Associations**

| | |
|---|---|
| source : Part [1] | Part that is the source of the Directed Part Connection<br>Subsets *connected part* |

| | |
|---|---|
| target : Part [1] | Part that is the target of the Directed Part Connection<br>Subsets *connected part* |

### 6.2.2.12 Generalization

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** "Element"

**Description**

A generalization between two types means each instance of the specific type is also an instance of the general type. Any specification applying to instances of the general type also apply to instances of the specific type.

**Associations**

| | |
|---|---|
| general : Type [1] | References the general Type in the Generalization relationship. |

### 6.2.2.13 Individual

**Namespace:** Composition Model
**isAbstract:** No

**Description**

Individual instance

### 6.2.2.14 Individual From Set

**Namespace:** Composition Model
**isAbstract:** No
**Generalization:** "Selector Specification"

**Description**

A **Individual From Set** is a kind of **Selector Specification** that provide a list of **Individual** as the potential **Type** of a **Typed Part**

**Associations**

| | |
|---|---|
| member : Individual [*] | Individual member of a Individual From Set selector specification |

### 6.2.2.15 Irreflexive Condition

**Namespace:** Composition Model
**isAbstract:**
**Generalization:** "Opaque Condition"

### Description

An **Irreflexive Condition** is a kind of **Opaque Condition** that restricts the connection to apply at M0 only to distinct M0 individuals playing the part (or playing the last part in the path). It applies only to connections between **Typed Parts**, or **Part Paths** with at least one **Typed Part**.

## 6.2.2.16 Opaque Condition

**Namespace:** Composition Model
**isAbstract:**
**Generalization:** "Condition"

### Description

An **Opaque Condition** is a **Condition** that enables the condition to be expressed textually in multiple languages.

### Associations

| | |
|---|---|
| condition specification : ValueSpecification [1] | ValueSpecification that specifies the expression of the Opaque Condition<br>Subsets *ownedElement* |

## 6.2.2.17 Part

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** "Abstract Part"

### Description

A **Part** is an element of the structure of a **Composite**. See **Abstract Part**

### Associations

| | |
|---|---|
| part path : Part Path [*] | Part Path that the part is traversed by |

| | |
|---|---|
| source connection : Directed Part Connection [*] | Directed Part Connection that the Part is the target of<br>Subsets *part connection* |

| | |
|---|---|
| target connection : Directed Part Connection [*] | Directed Part Connection that the part is the source of<br>Subsets *part connection* |

## 6.2.2.18 Part Connection

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** "Element"

### Description

A **Part Connection** is used to connect **Parts** of a **Composite**. A **Part Connection** can connect any number of parts. For example, a business interaction can involve multiple companies.

When a **Part Connection** is connecting **Typed Part**, its specifies links between M0 entities playing the typed parts. For example, the reporting connection between the president of a company and the CEO means the person playing the president in a particular company will report to the person playing the CEO in the same company. Likewise, the temporal connection between one step and another in a process means that in each occurrence of that process, there is an occurrence of one step that happens after the occurrence of another.

**Conditions** may be applied to **Part Connections** to limit when they apply. For example, one step in a process may happen after another only when certain conditions are true as the process is executing.

#### Associations

| | |
|---|---|
| connected part : Abstract Part [2..*] | Abstract Part connected by a Part Connection |

| | |
|---|---|
| guard : Condition [0..1] | Condition evaluated at runtime to determine if the Part Connection is enabled.<br><br>Subsets *ownedElement* |

### 6.2.2.19 Part Group

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** “Part”

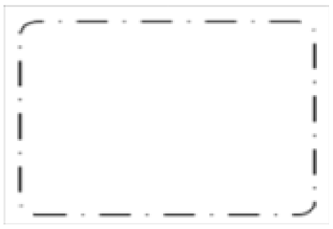
#### Description

A **Part Group** is a kind of **Part** that collects other **Part**s together. A **Part Groups** can share **Parts**. The meaning of part groups is given in the specializations of the **Composition Model**, for example, in **Processing Behavior**.

#### Associations

| | |
|---|---|
| enclosed part : Part [*] | Part that is enclosed in a Part Group. A Part can be enclosed in multiple Part Groups |

#### BPMN Notation



Part Group

**Figure 6.6 - Part Group Notation**

### 6.2.2.20 Part Path

**Namespace:** Composition Model

**isAbstract:** Yes
**Generalization:** “Abstract Part”

### Description

A **Part Path** connects to a **Part** of a nested **Composite**.

An instance of **Part Path** is introduced for each **traversed part** to a **target part** .

The purpose of **Part Path** is to provide access to parts in a nested composite structure. All models based on the composition model needs to have access to parts within parts, for example;

- Data elements within data elements
- Roles within roles
- Protocols within protocols
- Activities within activities

**Part Path** and a **Part** are generalized to **Abstract Part**, which are the is of **Part Connection**. This enables connections not requiring part paths to refer directly to parts, rather than to part paths with only one element.

### Associations

| | |
|---|---|
| final target : Part [1] | leaf Part to which a part path chain is pointing at |

| | |
|---|---|
| target part : Abstract Part [1] | Abstract Part to which the part path is pointing at. |

| | |
|---|---|
| traversed part : Part [1] | Part being the source of the part path. This part is traversed by the part path in order to reach the target part. |

### Constraint

[2] The **target part** must be a **Part** of the **Composite** that owns the **target part**

self.**target part** in self.**traversed part** ->**partType** ->**composite part**

[2] The **traversed part** must be a **Typed Part**

self.**traversed part** .isKindOf(**Typed Part**)

**Part Path**

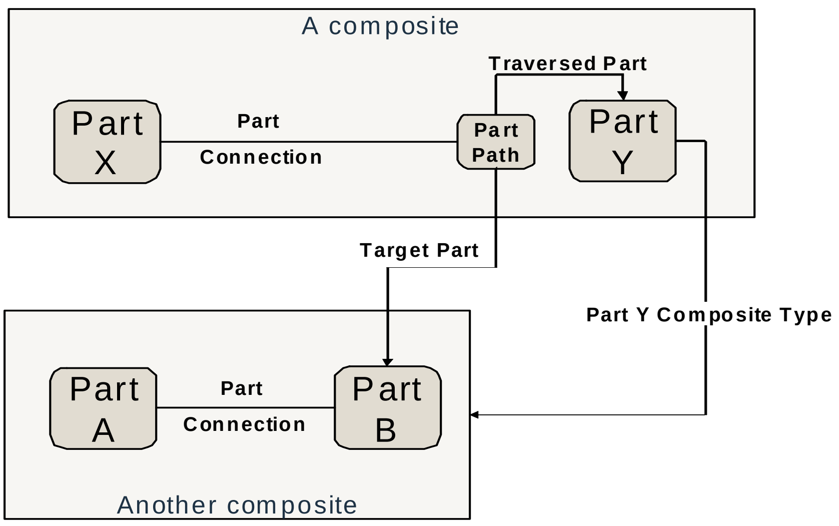


**Figure 6.7 - Part Path**

### 6.2.2.21 Part Replacement

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** "Element"

#### Description

A **Part Replacement** is used to specify the replacement or removal of **Part**s in **derived to Composite** of a **Derivation**.

#### Associations

| | |
|---|---|
| derived from : Part [*] | |

| | |
|---|---|
| derived to : Part [*] | |

### 6.2.2.22 Selector Specification

**Namespace:** Composition Model
**isAbstract:** Yes
**Generalization:** "ValueSpecification"

#### Description

A **Selector Specification** is a query mechanism used to specify the individuals playing a **Typed Part**

### 6.2.2.23 Typed Part

**Namespace:** Composition Model
**isAbstract:** Yes

**Generalization:** “Part” “TypedElement” “MultiplicityElement”

#### Description

A **Typed Part** is a kind of **Part** that specifies that individuals playing the **Part** in the **Composite** must be of a certain kind (**Type**). For example, an individual playing the president part must be a person, and must play the president within a individual company.

**Typed Part**] is a **MultiplicityElement**for restricting the number of individuals that play the part. For example, a company might allow no more than five vice-presidents, but require a president, and a choreography might have an interaction that is optional.

#### Associations

| | |
|---|---|
| partType : Type [1] | Type of the Typed Part<br>Subsets *type* |

| | |
|---|---|
| selection rule : Selector Specification [*] | Selector Specification used to specify the individual that plays the Typed Part<br>Subsets *ownedElement* |

### 6.2.2.24 Instance: Irreflexive Condition

**Class:** Irreflexive Condition

#### Description

This condition is applicable to connections between parts (or paths with at least one typed part). It restricts the connection to apply at M0 only to distinct M0 individuals playing the part (or the last part in the path).

#### Usage convention

A connection with one part typing a connector implies part bindings between the end parts of the connector and the single part of the connector type.

#### Links

| Played End | Opposite End |
|---|---|
| guard | finish/abort |

## 6.3 Course Model

### 6.3.1 Introduction

The Course Model extends the Composition Model to connect parts in time (Succession). For example, a succession connects one step in a process to another to indicate that the second step happens after the first. The same applies to messages in choreography.

Courses are Composites that have Succession connections representing that one part of the course "follows" another in time, and possibly establishes constraints on such followings. This can have very different meanings for typed and un-typed parts:

- For Typed Course Parts, Succession means that an individual dynamic entity playing one typed part will happen at the same time or after another dynamic entity playing another typed part as the course proceeds. These dynamic entities might be steps in a process, interactions in choreography, or changes and events due to these. Immediate Successions are Successions where the dynamic entities being connected happen at the same time. For example, two steps in a process might be required to start at the same time (see the Happening and Change Model).
- For un-typed course parts, such as Course Control Parts, Successions represent more complex specifications of how dynamic individuals playing typed parts are ordered in time. Parallel Splits are Course Control Parts indicating that the dynamic individuals playing parts following them happen after the dynamic individuals playing the part preceding them. Parallel Joins indicate that the parts (in the sense of individuals) following them happen after the parts preceding them. Exclusive Splits indicate that exactly one of the parts following them will occur after the part preceding them. Exclusive Joins indicate that the part following them will occur after each part that occurs preceding them. Successions with un-typed parts at one or both ends may not have part paths at those ends, including qualification, because there will be no individual playing that part (see Composition Model).

Course Parts are defined just to categorize those Parts that may be related by Successions.

### 6.3.2 Metamodel Specification

The Course Model extends the Composition Model to connect parts in time (Succession). For example, a succession connects one step in a process to another to indicate that the second step happens after the first. The same applies to messages in choreography.

### 6.3.2.1 Course Model

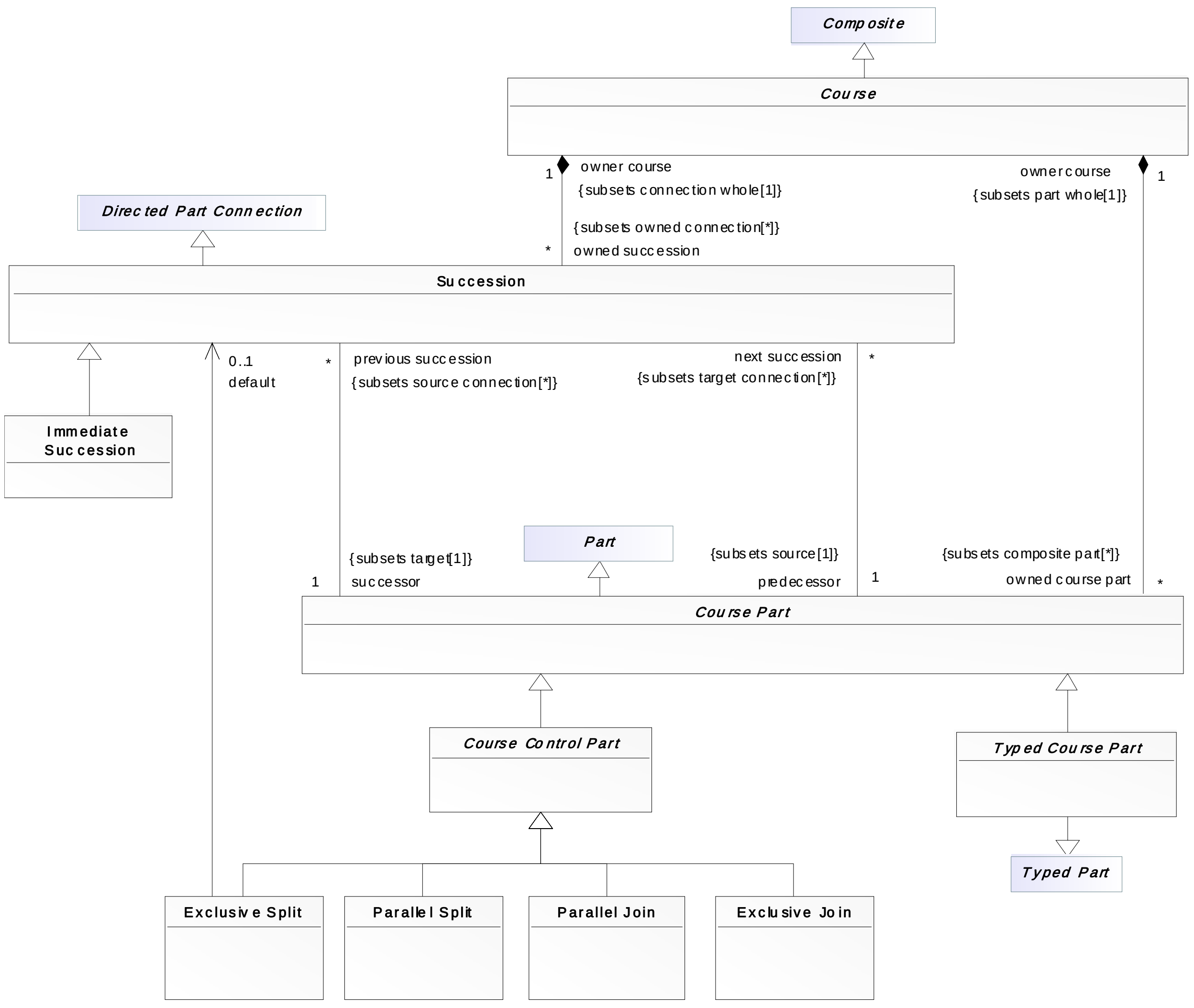


**Figure 6.8 - Course Model**

### 6.3.2.2 Course

**Namespace:** Course Model
**isAbstract:** Yes
**Generalization:** “Composite”

#### Description

A **Course** is an ordered **Succession** of **Typed Course Part**s

A **Course** is a **Composite** that has connections representing that one part of the course “follows” another in time, and possibly establishes constraints on such followings (**Succession**).

### Associations

| | |
|---|---|
| owned course part : Course Part [*] | Course Part owned by the Course<br>Subsets *composite part* |

| | |
|---|---|
| owned succession : Succession [*] | Succession owned by the Course<br>Subsets *owned connection* |

### 6.3.2.3 Course Control Part

**Namespace:** Course Model
**isAbstract:** Yes
**Generalization:** "Course Part"

#### Description

A **Course Control Part** is a kind of **Course Part** representing potentially complex specifications of how dynamic individuals playing **Typed Course Part**s are ordered in time. The particular specifications are given in subtypes. At runtime, **Course Parts** don't have any execution trace.

#### BPMN Notation

The shape of a Course Control Part is called a Gateway. A Gateway is a diamond which has been used in many flow chart notations for exclusive branching and is familiar to most modelers. The diamond MUST be drawn with a single thin black line.

It is not a requirement that predecessor and successor Successions must connect to the corners of the diamond. Successions can connect to any position on the boundary of the Gateway.

The shape of the different sub-types of Course Control Part are differentiated by an internal marker. This marker MUST be placed inside the shape, in any size or location, depending on the preference of the modeler or modeling tool vendor.

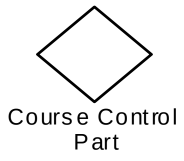


**Figure 6.9 - Course Control Part Notation**

### 6.3.2.4 Course Part

**Namespace:** Course Model
**isAbstract:** Yes
**Generalization:** "Part"

#### Description

A **Course Part** is a kind of **Part** that defines a stage in a **Course**.

It can be connected to **Succession** as a **predecessor** or **successor** element.

### Associations

| | |
|---|---|
| next succession : Succession [*] | Succession that enables the Course Part as its predecessor<br>Subsets *target connection* |

| | |
|---|---|
| previous succession : Succession [*] | Succession that enables the Course Part as its successor<br>Subsets *source connection* |

## 6.3.2.5 Exclusive Join

**Namespace:** Course Model
**isAbstract:** No
**Generalization:** "Course Control Part"

### Description

An **Exclusive Join** is a **Course Control Part** indicating that the part following it will occur after each part that occurs preceding it.

### BPMN Notation

The Exclusive Join shares the same basic shape of the generic **Course Control Part**.

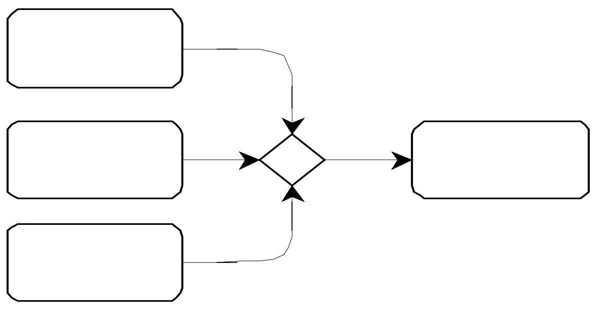


Figure 6.10 - Exclusive Merge Notation

## 6.3.2.6 Exclusive Split

**Namespace:** Course Model
**isAbstract:** No
**Generalization:** "Course Control Part"

### Description

**Exclusive Split** is a **Course Control Part** indicating that exactly one of the part following it will occur after the part preceding it.

### Associations

| | |
|---|---|
| default : Succession [0..1] | Succession enabled by default if no other next succession connected to the Exclusive Split has been enabled. |

### Constraint

[1] The **default Succession** must be one of the **Successions** connected to the **Exclusive Split** as a **next succession**

### BPMN Notation

The Exclusive Split shares the same basic shape, called a Gateway,  of the generic **Course Control Part**. The Exclusive Split MAY use a marker that is shaped like an “X” and is placed within the Gateway diamond to distinguish it from other **Course Control Parts**. This marker is not required . A Diagram SHOULD be consistent in the use of the “X” internal indicator. That is, a Diagram SHOULD NOT have some Exclusive Splits with an indicator and some Exclusive Splits without an indicator.

The **default** succession is represented by a default Marker that MUST be a backslash near the beginning of the line representing the **Succession**.

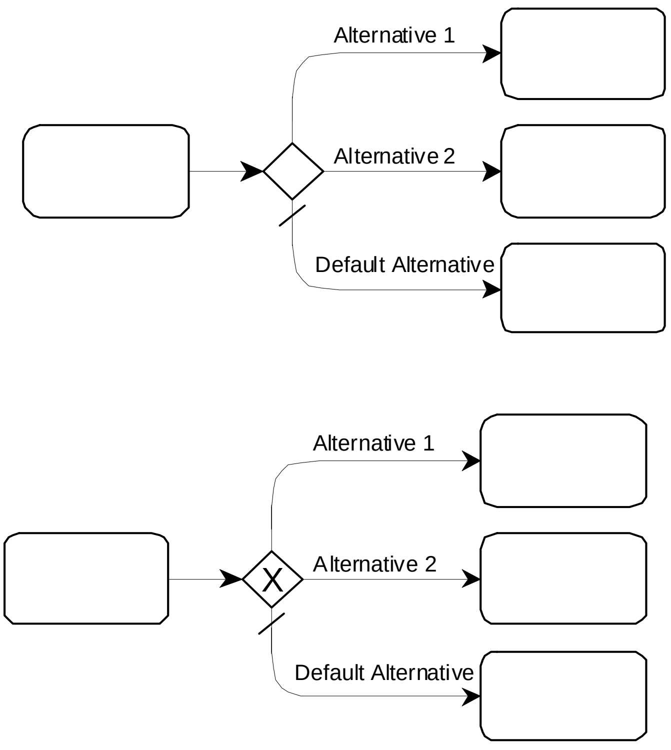


**Figure 6.11 - Exclusive Split Notation**

### 6.3.2.7 Immediate Succession

**Namespace:** Course Model
**isAbstract:**
**Generalization:** “Succession”

#### Description

A **Immediate Succession** is a kind of **Succession** that has the following execution semantic: **successor** immediately follows its **predecessor**

### 6.3.2.8 Parallel Join

**Namespace:** Course Model
**isAbstract:** No
**Generalization:** “Course Control Part”

#### Description

**Parallel Join** is a **Course Control Part** indicating that the parts (in the sense of individuals) following it happen after the parts preceding them.

#### BPMN Notation

The Parallel Join uses the shape of **Course Control Part**, called Gateway and MUST use a marker that is in the shape of an plus sign and is placed within the Gateway diamond to distinguish it from other of **Course Control Parts.**

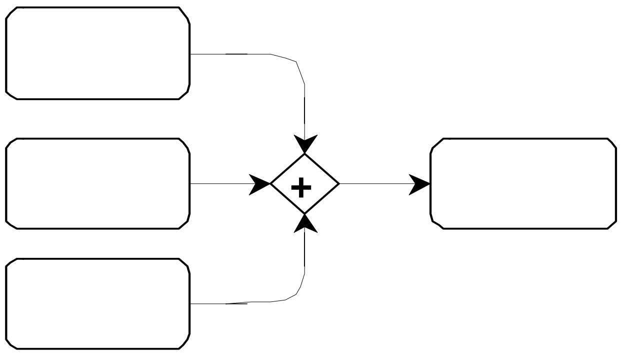


**Figure 6.12 - Parallel Join Notation**

### 6.3.2.9 Parallel Split

**Namespace:** Course Model
**isAbstract:** No
**Generalization:** “Course Control Part”

#### Description

**Parallel Split** is a **Course Control Part** that indicates that the dynamic individuals playing parts following them happen after the dynamic individuals playing the part preceding them.

### BPMN Notation

The Parallel Split uses the shape of **Course Control Part**, called Gateway and MUST use a marker that is in the shape of an plus sign and is placed within the Gateway diamond to distinguish it from other of **Course Control Parts**.

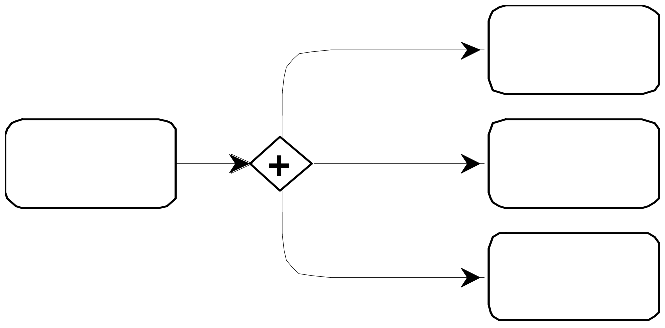


**Figure 6.13 - Parallel Split Notation**

#### 6.3.2.10 Succession

**Namespace:** Course Model
**isAbstract:** No
**Generalization:** "Directed Part Connection"

### Description

A **Succession** is a **Directed Part Connection** that organizes **Course Parts** in series in the context of a **Course**. A **Succession** indicates that  that one **Course Part** "follows" another in time, and possibly establishes constraints on such followings.

### Associations

| | |
|---|---|
| predecessor : Course Part [1] | Course Part that comes before another Course Part in a Succession<br>Subsets *source* |

| | |
|---|---|
| successor : Course Part [1] | Course Part that comes after another Course Part in a Succession<br>Subsets *target* |

### BPMN Notation

A Succession is line with a solid arrowhead that MUST be drawn with a solid single line

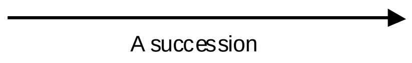


**Figure 6.14 - Succession Notation**

#### Non-normative Notation

A Succession with a **Condition** of type **Fact Change Condition**is drawn as a line covered by the shape the conditioning **Fact Change**.

The line has a solid arrowhead and MUST be drawn with as solid single line.

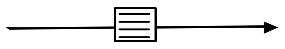


**Figure 6.15 - Succession with Fact Change Condition**

A  Succession with a **Condition** of type **Time Change Condition** is drawn as one line covered by the shape the conditioning **Time Change**.

The line has a solid arrowhead and MUST be drawn with as solid single line

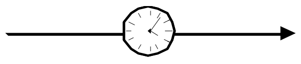


**Figure 6.16 - Succession with Time Change Condition**

### 6.3.2.11  Typed Course Part

**Namespace:** Course Model
**isAbstract:** Yes
**Generalization:** “Course Part” “Typed Part”

#### Description

A **Typed Course Part** is kind of **Course Part** that is a stage or interval in a development or **Course**.

**Typed Course Part**s are  different from other **Course Part**s as they are the only one that have occurence trace at runtime.

## 6.4 Happening & Change Model

### 6.4.1 Introduction

The Happening and Change Model introduces dynamics, in particular, time ordering of lifecycle events, such as starting and ending a process. This facilitates the integration of rule and monitoring systems with models of dynamics, such as orchestration and choreography.  The model enables users and vendors to define their own libraries of processes, with their own categorizations and attributes, such as how long a process has been running, and the resources it is using. They can also define their own life cycle events, for example, to define finish statuses and taxonomy of errors.

The Happening and Change Model extends the Composition and Course Models with:

- General categories for dynamic entities that extend over time (Happenings Over Time) producing entities that occur at a point in time (Changes).

- Courses that produce lifecycle events, such as starting and ending, enabling the events to be ordered in time (Behavioral Happenings and Behavioral Changes).
- A user (M1) library defining a behavioral happening that produces common behavior lifecycle events, such starting and ending (Universal Behavioral Happening).
- Conditions for time changes and changes in facts.

Happenings Over Time and Changes are Types for dynamic entities that are treated as extending over time, or as occurring at a point in time, respectively. Happenings over time produce changes, for example, the revenue of a company changes during a business process. An individual dynamic entity could be either a happening over time or a change, depending on the viewpoint of the application.  For example, a package arriving at a business might be treated as a process of signing for it, inspecting it, and routing it to the addressee, or it might be treated as simply occurring on a particular day with no additional detail.

Behavioral Happenings are Happenings Over Time that produce Behavioral Changes, which are behavior lifecycle events, such as starting and ending. Behavioral Happenings are also Courses, enabling their lifecycle changes to be ordered in time by Successions (see the Course Model). Behavioral Change Parts identify behavioral changes for individual Behavioral Happenings. For example, a behavioral change part for shipping a product can identify the starting change for each individual shipment, such as 8am on a particular day. Behavioral Change Parts are also Course Parts, enabling them to be connected by Successions.  For example, a change part identifying the end of a behavioral happening succeeds the change part identifying the start. This means the ending of each individual behavioral happening, such as an individual shipment, is after the start of that same individual happening.

A user (M1) library in the Happening and Change Model captures commonly needed aspects of behavioral happenings as instances of classes in the Happening and Change Model. The library defines:

- Behavioral Changes to represent various behavior lifecycle events, such as starting and ending of individual behavioral happenings.
- A Behavioral Happening called the Universal Behavioral Happening. It is a generalization of all M1 dynamic models (see the Composition Model).
- Behavioral Change Parts of the Universal Behavioral Happening for the various Behavioral Changes, such as startPart and endPart. These are typed by the various M1 changes, such as Start and End Behavioral Changes.
- Successions between the Behavior Change Parts above for universal constraints, such as the end being after the start.

The library enables Users and extenders of BPDM to define their own:

- Parts on happenings, for example, a business monitoring model or business runtime model can specialize the Universal Behavioral Happening to introduce typed parts for the time an individual process starts, how long it has been running, and the resources it is using.
- Taxonomies of happenings, for example, a general business process can be specialized for small and large businesses, or business in specific sectors, such as health care or retail. This can be the framework for libraries of reusable business processes.
- Taxonomies of changes, for example, to define kinds of errors and introduce error codes.
- Behavioral change parts, for example, multiple finish parts for different finish statuses (success, failure, for example).[7]

7. For some individual (M0) happenings, the finish change will play the part of successful finish, for others it will play the part of failed finish.

These can be the source for successions leading to different steps taken as the result of finishing status.

The most general Behavior Changes in the user library are for the starting and ending of happenings (Start and End). End changes at M1 specialize into changes for aborting, erroring, and finishing, no two of which can occur in the same individual happening.  Aborting means the happening is terminated by an external source for abnormal reasons. Erroring means the happening is terminated by itself for abnormal reasons. Aborting and erroring may involve cleanup, but this must be completed before the end of the happening. Finishing means the happening ended normally, without aborting or erroring.[8]

Individual (M0) changes conforming to the library must play the library behavioral change parts as the changes occur.[9] For example, every individual conforming to the Universal Behavioral Happening will have an individual conforming to the library Start change that plays the value of its library startPart. Each individual (M0) universal behavioral happening will have at most one individual change conforming to the change types in the library. For example, there is only one start change for each individual universal behavioral happening. Inversely, each individual change must play a behavioral change part in exactly one individual universal behavioral happening.  For example, an M0 start change plays the startPart for exactly one individual universal behavioral happening.

Successions in Universal Behavioral Happening inherit to all user-defined behavior definitions (M1) and all individual (M0) behavioral happenings (all performances, enactments, and executions). These establish the time order of the lifecycle changes, for example that ending happens after starting. Successions that target parts typed by the Start change specify a new individual (M0) behavioral happening. For example, a process definition may indicate that an incoming message creates a new execution of a process by a succession from the message receipt to the start part in the user library (see the BPMN Extensions package).

Change Conditions are Conditions for specifying that a Change must occur in the context of a particular Happening Over Time for the condition to hold. It is specialized into conditions for Behavioral Changes, Time Changes, and changes in Facts. Behavioral Conditions specify that an individual (M0) behavioral happening must produce a particular kind of behavioral change (defined at M1) for the condition to hold. For example, a selling process may notify customers that products have been shipped, and the notification should happen only after the shipping step is complete. Each individual behavioral happening, such as each shipment, is identified by a Behavior Part (Process Steps are behavior parts, see the Processesing Behavior Model). Time Change Condition is specified by referring to a Clock, which is a Happening Over Time that produces Time Changes. Time Changes have a property for specifying the time in a detailed expression. Fact Change Conditions refer to general propositions becoming true or false due to changes in M0 facts. It is used to integrate with models of rules. All change conditions can identify a happening that produces the change.

### 6.4.2 Metamodel Specification

The Happening and Change Model introduces dynamics, in particular, time ordering of lifecycle events, such as starting and ending a process. This facilitates the integration of rule and monitoring systems with models of dynamics, such as orchestration and choreography.  The model enables users and vendors to define their own libraries of processes, with their own categorizations and attributes, such as how long a process has been running, and the resources it is using. They can also define their own life cycle events, for example, to define finish statuses and taxonomy of errors.

8. Finishing includes situations where the happening ends normally, but does not have the effect intended by the modeler.  See previous footnote and example.
9. See comments on conformance in the first footnote in Individuals, Models, and Modeling Languages in the Composition Model.

### 6.4.2.1 Happening & Change

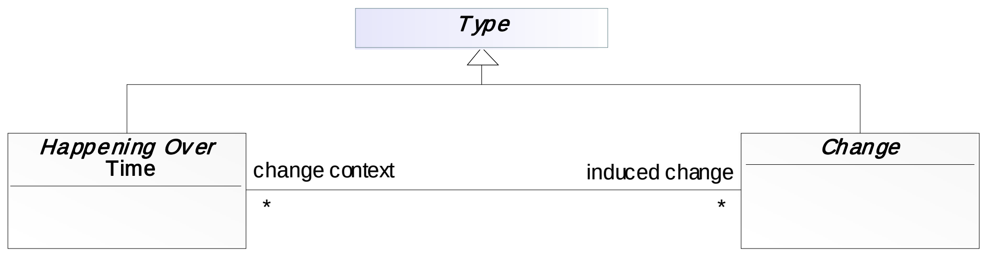


**Figure 6.17 - Happening & Change**

### 6.4.2.2 Behavioral Happening

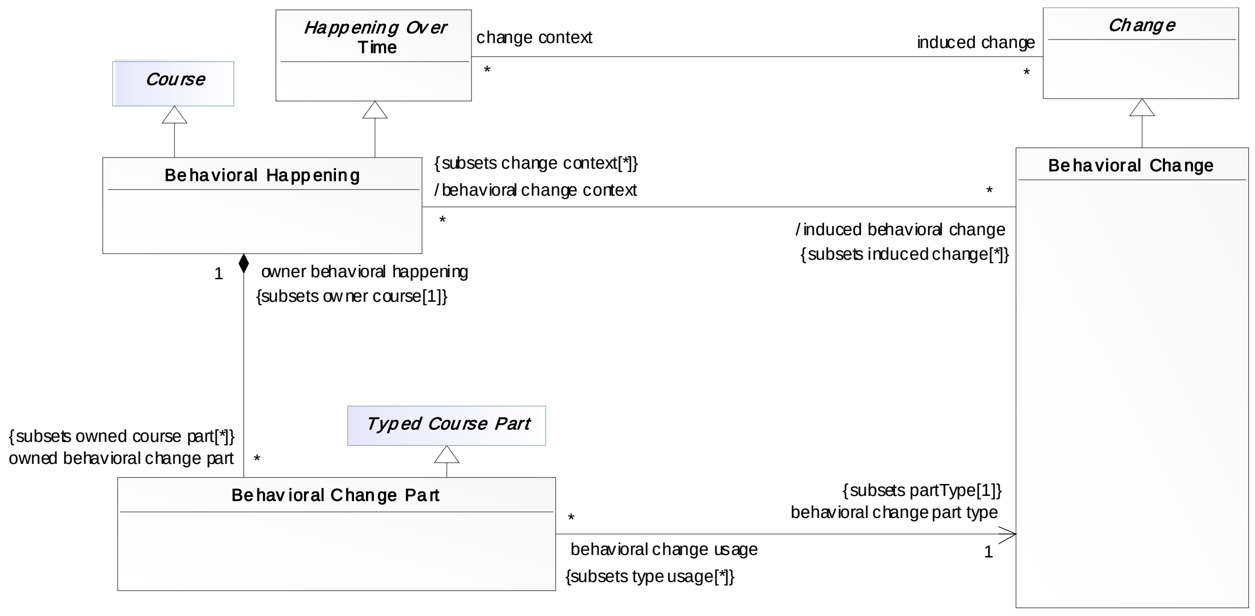

Course
Happening Over Time
change context
*
induced change
*
Change
Behavioral Happening
{subsets change context[*]}
/behavioral change context
*
*
/induced behavioral change
{subsets induced change[*]}
Behavioral Change
1
owner behavioral happening
{subsets owner course[1]}
{subsets owned course part[*]}
owned behavioral change part
*
Typed Course Part
Behavioral Change Part
*
behavioral change usage
{subsets type usage[*]}
{subsets partType[1]}
behavioral change part type
1

**Figure 6.18 - Behavioral Happening**

### 6.4.2.3 Happening & Change Library: Behavioral Change instances

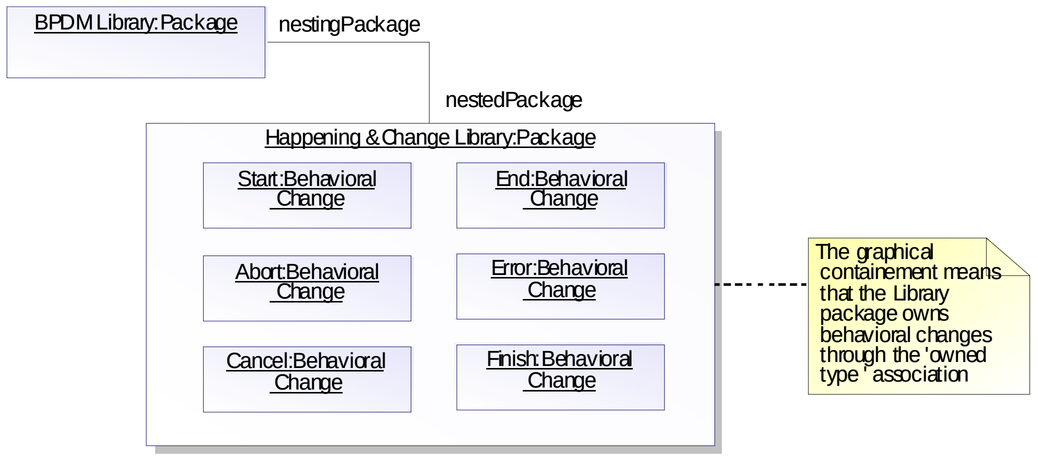


**Figure 6.19 - Happening & Change Library: Behavioral Change instances**

### 6.4.2.4 Happening & Change Library: 'Universal Behavioral Happening' instance

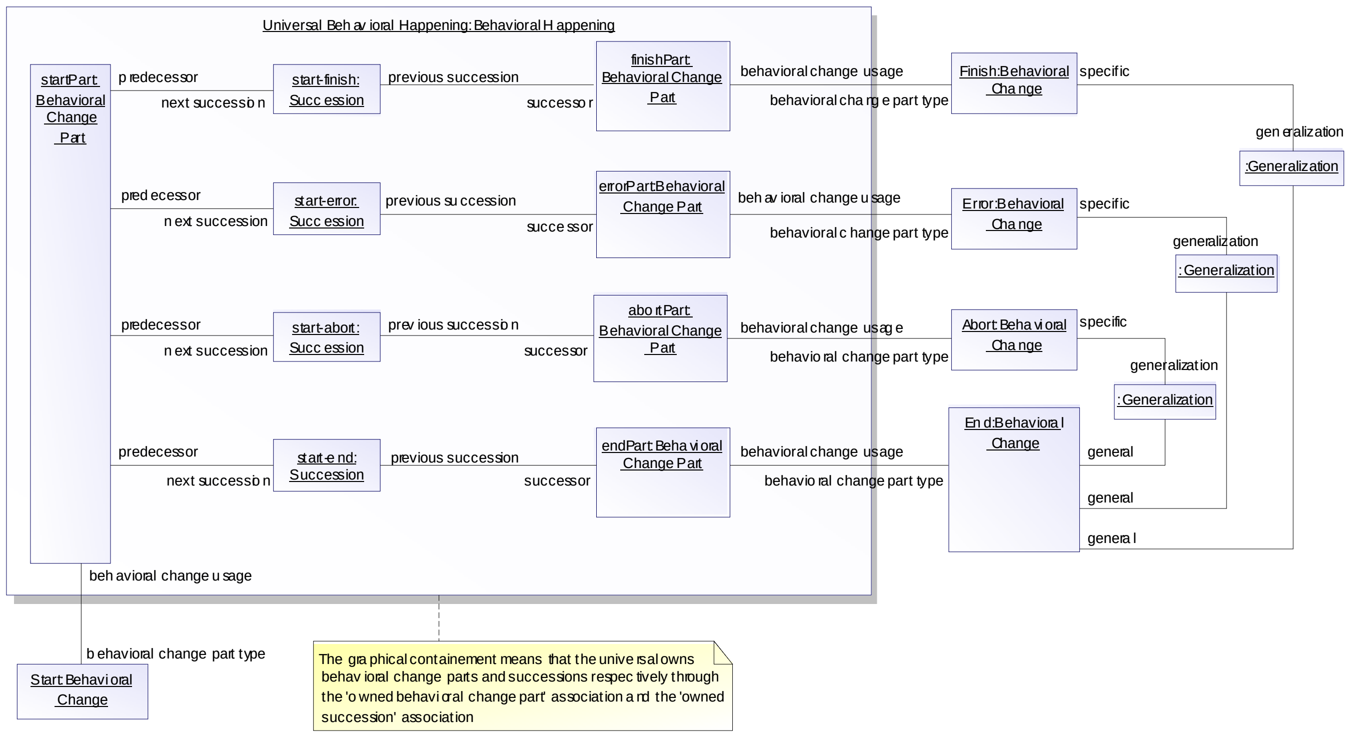

Universal Behavioral Happening:Behavioral Happening
startPart: Behavioral Change Part
predecessor
next succession
start-finish: Succession
previous succession
successor
finishPart: Behavioral Change Part
behavioral change usage
behavioral change part type
Finish:Behavioral Change
specific
generalization
:Generalization
start-error: Succession
errorPart:Behavioral Change Part
Error:Behavioral Change
start-abort: Succession
abortPart: Behavioral Change Part
Abort:Behavioral Change
start-end: Succession
endPart: Behavioral Change Part
End:Behavioral Change
general
Start:Behavioral Change
The graphical containement means that the universal owns behavioral change parts and successions respectively through the 'owned behavioral change part' association and the 'owned succession' association

**Figure 6.20 - Happening & Change Library: 'Universal Behavioral Happening' instance**

### 6.4.2.5 Change Condition

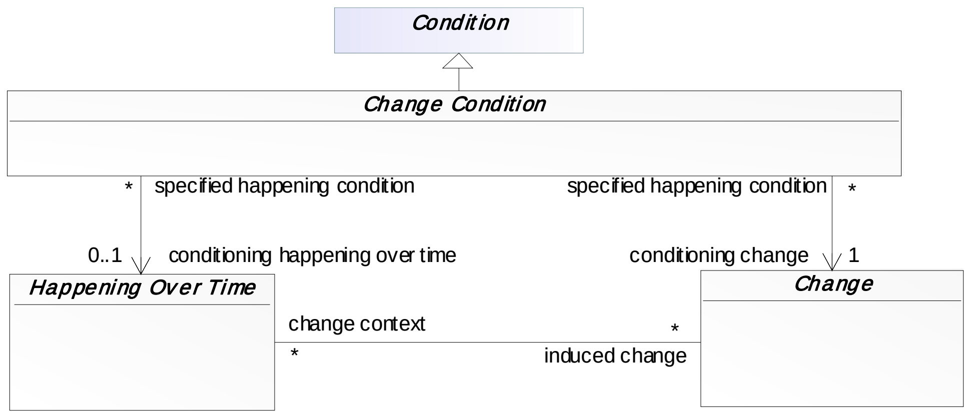
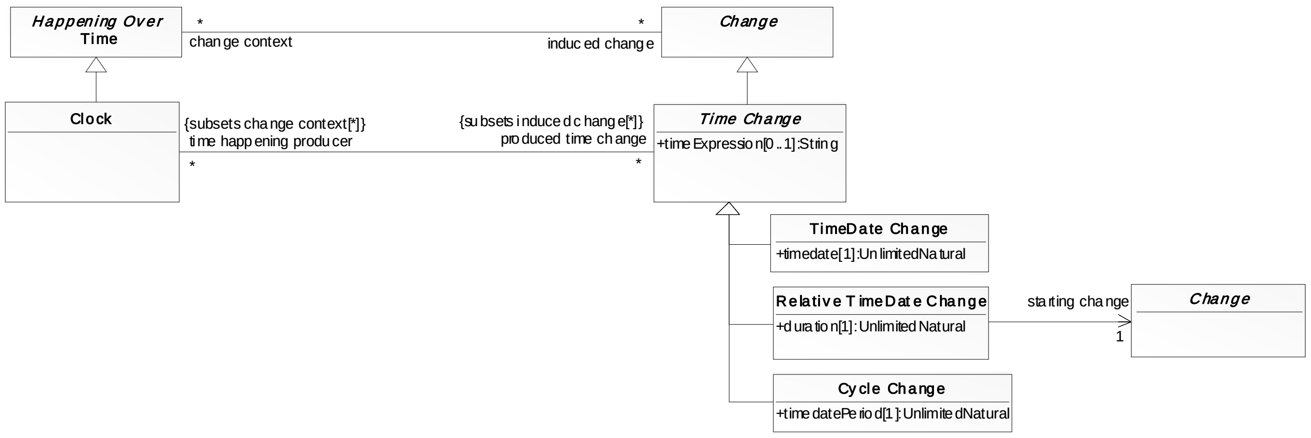


**Figure 6.21 - Change Condition**

### 6.4.2.6 Time Change



Happening Over Time
* change context
* induced change
Change
Clock
{subsets change context[*]} time happening producer
*
{subsets induced change[*]} produced time change
*
Time Change
+timeExpression[0..1]:String
TimeDate Change
+timedate[1]:UnlimitedNatural
Relative TimeDate Change
+duration[1]:UnlimitedNatural
starting change
1
Change
Cycle Change
+timedatePeriod[1]:UnlimitedNatural

**Figure 6.22 - Time Change**

### 6.4.2.7 Time Change Condition

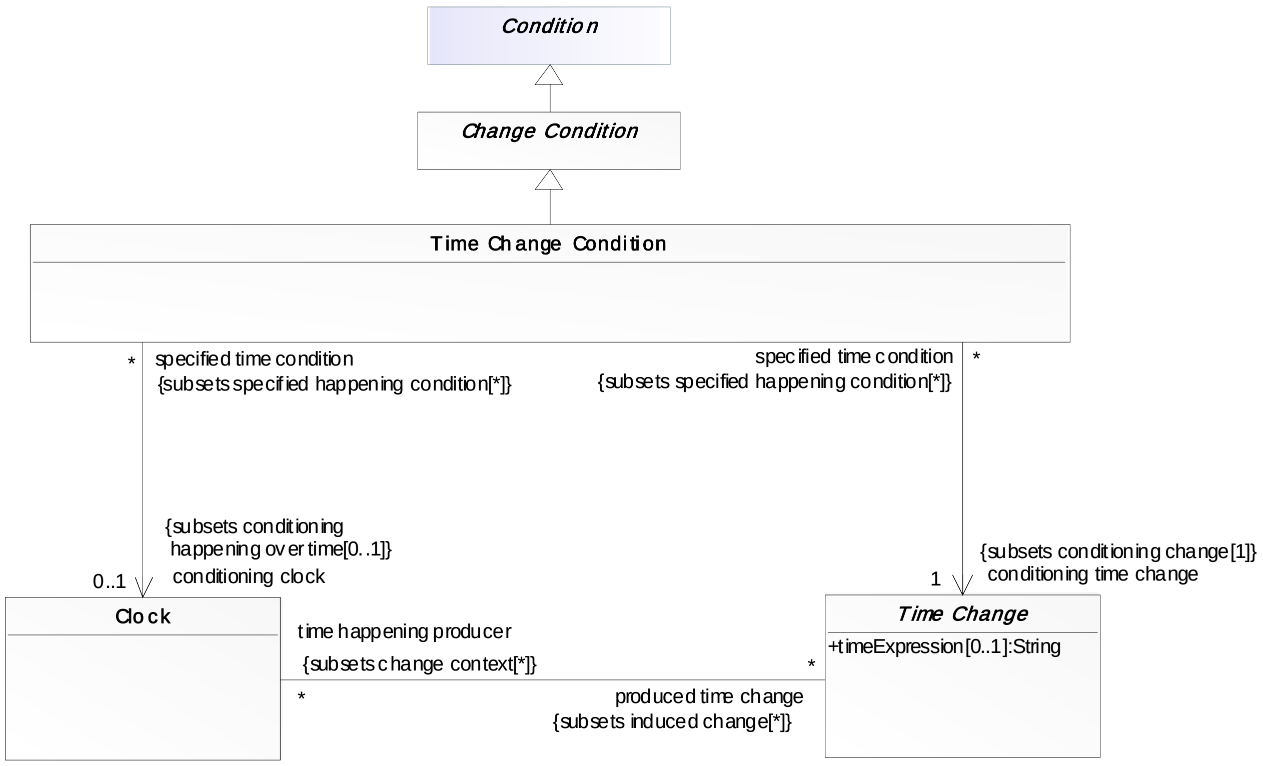

Condition
Change Condition
Time Change Condition
*
specified time condition
{subsets specified happening condition[*]}
specified time condition
{subsets specified happening condition[*]}
*
{subsets conditioning happening over time[0..1]}
conditioning clock
0..1
{subsets conditioning change[1]}
conditioning time change
1
Clock
time happening producer
{subsets change context[*]}
*
*
produced time change
{subsets induced change[*]}
Time Change
+timeExpression[0..1]:String

**Figure 6.23 - Time Change Condition**

### 6.4.2.8 Happening & Change Library : Fact Change instances

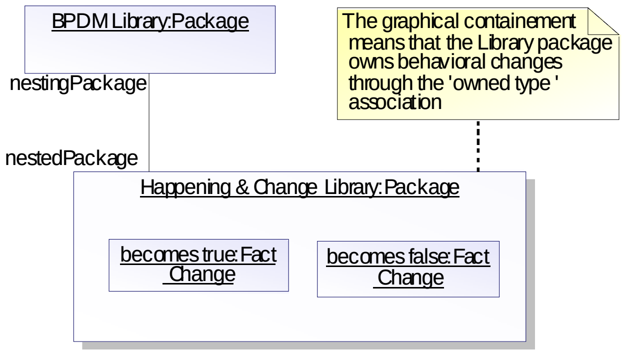

BPDM Library:Package
nestingPackage
nestedPackage
The graphical containement means that the Library package owns behavioral changes through the 'owned type ' association
Happening & Change Library:Package
becomes true:Fact Change
becomes false:Fact Change

**Figure 6.24 - Happening & Change Library : Fact Change instances**

### 6.4.2.9 Fact Change Condition

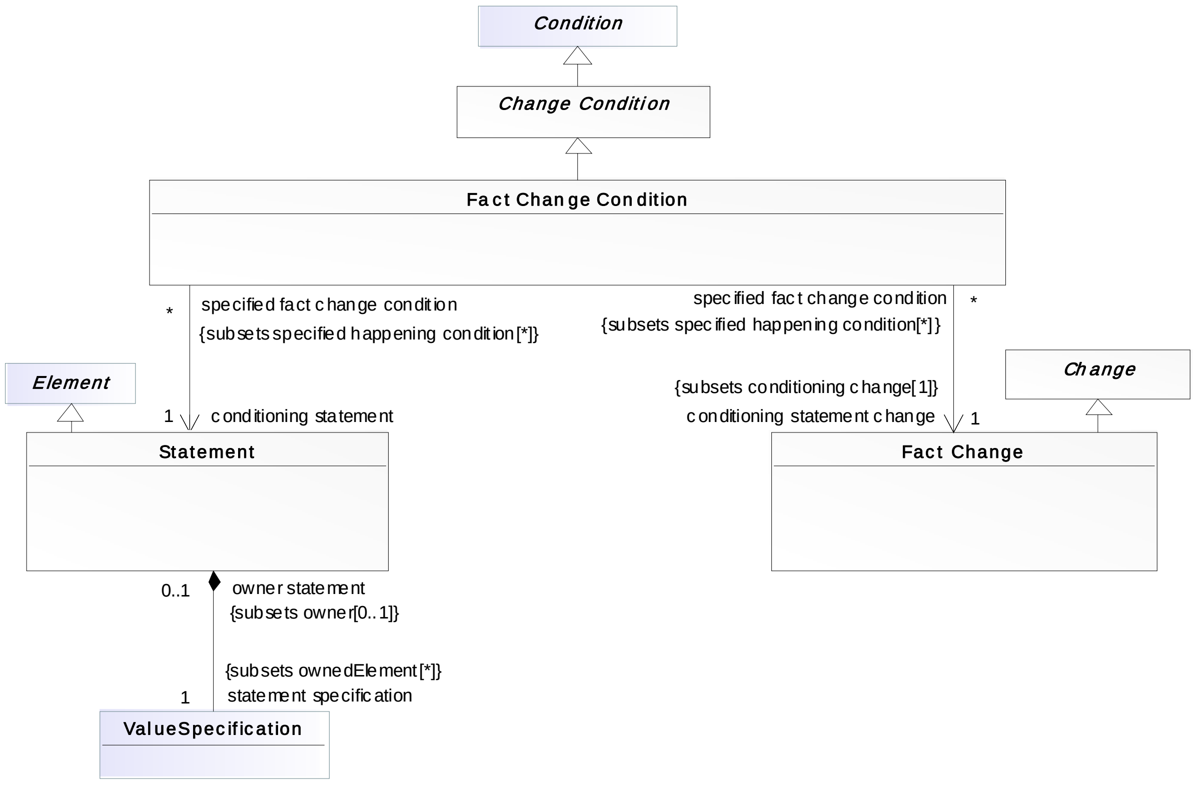

Condition
Change Condition
Fact Change Condition
*
specified fact change condition
{subsets specified happening condition[*]}
specified fact change condition
{subsets specified happening condition[*]}
*
Element
Change
1
conditioning statement
{subsets conditioning change[1]}
conditioning statement change
1
Statement
Fact Change
0..1
owner statement
{subsets owner[0..1]}
{subsets ownedElement[*]}
statement specification
1
ValueSpecification

**Figure 6.25 - Fact Change Condition**

### 6.4.2.10 Behavioral Change

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Change”

#### Description

A **Behavioral Change** is a kind of **Change** that occurs as part of the lifecycle of a **Behavioral Happening**, such as **Start**, **Finish** or **Abort**.

BPDM provides a predefined library of **Behavioral Changes**.

#### Associations

| | |
|---|---|
| behavioral change context : Behavioral Happening [*] | Behavioral Change that can occur in the context of the Behavioral Happening<br>Subsets *change context* |

### 6.4.2.11 Behavioral Change Part

**Namespace:** Happening & Change
**isAbstract:**
**Generalization:** "Typed Course Part"

#### Description

A **Behavioral Change Part** identifies **Behavioral Change** (such as **Start** or **End**) for an individual **Behavioral Happening**. A **Behavioral Change Part** is also a **Course Part**, enabling it to be connected by **Successions**.

#### Associations

| | |
|---|---|
| behavioral change part type : Behavioral Change [1] | Behavioral Change that is the type of the Behavioral Change Part<br>Subsets *partType* |

#### BPMN Notation

**Behavioral Change Part** typed by the **Cancel** instance of **Behavioral Change**

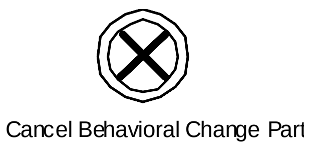


**Figure 6.26 - Behavioral Change Part : Cancel Notation**

A **Behavioral Change Part** typed by the **End** instance of **Behavioral Change** is drawn as a circle that MUST be drawn with a single thick black line.

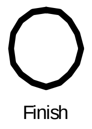


**Figure 6.27 - Behavioral Change Part : End Notation**

A **Behavioral Change Part** typed by the **Start** instance of **Behavioral Change** is drawn as a circle that MUST be drawn with a single thin line.

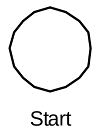


**Figure 6.28 - Behavioral Change Part : Start Notation**

When a **Start Behavioral Change Part** is conditionned by a **Fact Change Condition**, a **Fact Change** marker is added to the **Start Behavioral Change Part** shape.

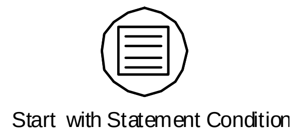


**Figure 6.29 - Behavioral Change Part : Start with 'Fact Change Condition' Notation**

When a **Start Behavioral Change Part** is conditionned by a **Time Change Condition**, a **Time Change** marker is added to the **Start Behavioral Change Part** shape.

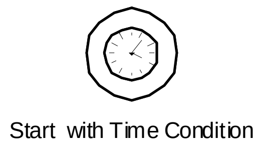


**Figure 6.30 - Behavioral Change Part : Start with 'Time Change Condition' Notation**

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Cancel** instance of **Behavioral Change**
2. A **Cancel Activity**

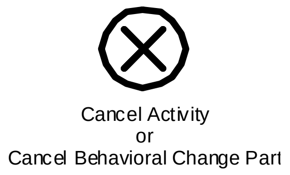


**Figure 6.31 - Cancel Activity Notation or 'Cancel' Behavioral Change Part**

This symbol is a a circle, with an open center. The circle MUST be drawn with a double thin black line. It can alternatively represent:

1. **Behavioral Change Parts** that are not typed by **Start** or **End**
2. **Change Condition Steps**

Markers can be placed within the circle to indicate the nature of the **Change** associated with the **Behavioral Change Part** or **Change Condition Step**

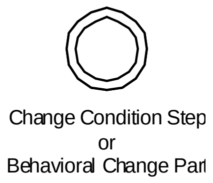


**Figure 6.32 - Change Condition Step Notation or Behavioral Change Part**

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Error** instance of **Behavioral Change**
2. An **Error Activity**

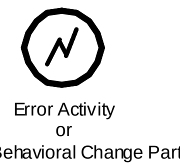


Figure 6.33 - Error Activity Notation or 'Error' Behavioral Change Part

**Error Behavioral Change Part** used for error handling.

The **Error Behavioral Change Part** is linked to the **Processing Succession** instance through the **predecessor behavioral change part** association.

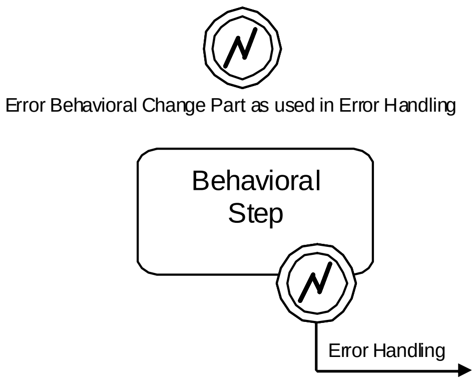


Figure 6.34 - Error Handling Notation

**Non Normative Notation**

A **Behavioral Change Part** typed by a **Finish** instance of **Behavioral Change** is drawn as a circle that MUST be drawn with a single thick black line.

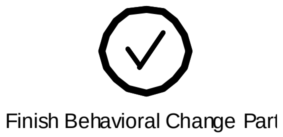


Figure 6.35 - Behavioral Change Part : Finish notation

### 6.4.2.12 Behavioral Happening

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Course” “Happening Over Time”

### Description

A **Behavioral Happening** is a kind of **Happening Over Time**that produces **Behavioral Changes** which are behavior lifecycle events, such as **Start** and **End**. A **Behavioral Happening** is also a **Course**, enabling its lifecycle to be ordered in time by **Successions** .

A user (M1) library - **Happening & Change Library** - captures commonly needed aspects of **Behavioral Happenings** as instances such as the finish being after the start.

### Associations

| | |
|---|---|
| induced behavioral change : Behavioral Change [*] | Behavioral Changes that can occur in the context of this Behavioral Happening.<br>The set of these Behavioral Changes is derived from the Behavioral Change Part owned by the Behavioral Happening.<br>Subsets *induced change* |

| | |
|---|---|
| owned behavioral change part : Behavioral Change Part [*] | Behavioral Change Part owned byt the Behavioral Happening<br>Subsets *owned course part* |

### Universal Behavioral Happening

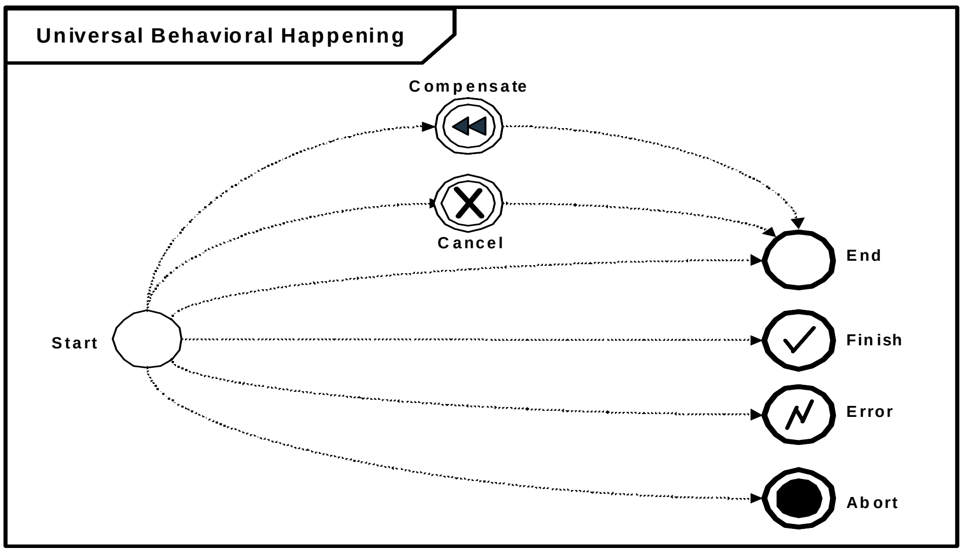


Figure 6.36 - Universal Behavioral Happening

## 6.4.2.13 Change

**Namespace:** Happening & Change
**isAbstract:** Yes
**Generalization:** “Type”

### Description

A **Change** is a **Type** for dynamic entities occurring at a point in time.

### Associations

| | |
|---|---|
| change context : Happening Over Time [*] | Happening Over Time where the Change can occur |

### 6.4.2.14 Change Condition

**Namespace:** Happening & Change
**isAbstract:** Yes
**Generalization:** “Condition”

### Description

A **Change Condition** is a **Condition** for specifying that a **Change** must occur in the context of a particular **Happening Over Time** for the condition to hold.

For instance, a condition can be on the eruption (instance of **Change**) of a particular volcano (instance of **Happening Over Time**).

### Associations

| | |
|---|---|
| conditioning change : Change [1] | Change that is the source of the Change Condition |

| | |
|---|---|
| conditioning happening over time : Happening Over Time [0..1] | Happening Over Time where the conditioning change should occur |

### 6.4.2.15 Clock

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Happening Over Time”

### Description

A **Clock** is a kind of **Happening Over Time** that produces **Time Changes**.

### Associations

| | |
|---|---|
| produced time change : Time Change [*] | Time Change that occurs in the context of a Clock<br>Subsets *induced change* |

### 6.4.2.16 Cycle Change

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Time Change”

### Description

A **Cycle Change** is a kind of **Time Change** that define the occurrence of a cycle in time.

### Attributes

| | |
|---|---|
| timedatePeriod: UnlimitedNatural [1] | |

### 6.4.2.17 Fact Change

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Change”

### Description

A **Fact Change** is a kind of **Change** that manifests a change in the evaluation of a **Statement**.

### BPMN Notation

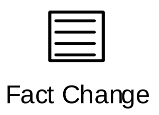


**Figure 6.37 - Fact Change Notation**

### 6.4.2.18 Fact Change Condition

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Change Condition”

### Description

A **Fact Change Condition** refers to general propositions becoming true or false due to changes in M0 facts. It is used to integrate with models of rules.

### Associations

| | |
|---|---|
| conditioning statement change : Fact Change [1] | Fact Change that, when it occurs, make the Fact Change Condition evaluate to true<br>Subsets *conditioning change* |

| | |
|---|---|
| conditioning statement : Statement [1] | Statement that the Fact Change Condition is evaluating the change of. |

### 6.4.2.19 Happening Over Time

**Namespace:** Happening & Change
**isAbstract:** Yes

**Generalization:** “Type”

### Description

A **Happening Over Time** is a **Type** for dynamic entities that are treated as extending over time and that are contexts for **Changes**.

### Associations

| | |
|---|---|
| induced change : Change [*] | Change that occurs in the context of the Happening Over Time |

### 6.4.2.20 Relative TimeDate Change

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Time Change”

### Description

A **Relative TimeDate Change** is a kind of **Relative TimeDate Change** that defines a change in time for a relative start point in time.

### Attributes

| | |
|---|---|
| duration: UnlimitedNatural [1] | |

### Associations

| | |
|---|---|
| starting change : Change [1] | Change which occurrence is the beginning of the Relative TimeDate Change |

### 6.4.2.21 Statement

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Element”

### Description

Representation of a proposition by an expression of the proposition

### Associations

| | |
|---|---|
| statement specification : ValueSpecification [1] | specification of the Statement<br>Subsets *ownedElement* |

### 6.4.2.22 Time Change

**Namespace:** Happening & Change
**isAbstract:** Yes
**Generalization:** “Change”

### Description

A **Time Change** specifies a point in time that is a source of interest.

### Attributes

| | |
|---|---|
| timeExpression: String [0..1] | A timeExpression represents a time value. |

### Associations

| | |
|---|---|
| time happening producer : Clock [*] | Clock that generates the Time Change<br>Subsets *change context* |

### BPMN Notation

A **Time Change** is represented by a clock

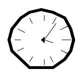


**Figure 6.38 - Time Change Notation**

## 6.4.2.23 Time Change Condition

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Change Condition”

### Description

A **Time Change Condition** is a kind of **Change Condition** that is based on the occurrence of a **Time Change**. A **Time Change Condition** is specified by referring to a **Clock**.

### Associations

| | |
|---|---|
| conditioning clock : Clock [0..1] | Clock that is the Happening Over Time context producing the conditioning time change that is the source of the Time Change Condition<br>Subsets *conditioning happening over time* |

| | |
|---|---|
| conditioning time change : Time Change [1] | Time Change that is the source of the Time Change Condition<br>Subsets *conditioning change* |

## 6.4.2.24 TimeDate Change

**Namespace:** Happening & Change
**isAbstract:** No
**Generalization:** “Time Change”

#### Description

A **TimeDate Change** is a kind of **Time Change** that manifest a date or time change

#### Attributes

| | |
|---|---|
| timedate: UnlimitedNatural [1] | |

### 6.4.2.25 Instance: Abort

**Class:** Behavioral Change

#### Description

**Abort** is a **Behavioral Change** that manifests that the the course of a **Behavioral Happening** is being interrupted. The source of the **Abort** can be internal or external to the **Behavioral Happening**.

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* abortPart |
| induced behavioral change | *behavioral change context* Abort Process |
| ownedType | *package* Happening & Change Library |
| specific | *generalization* Generalization |

### 6.4.2.26 Instance: abortPart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Abort |
| owned behavioral change part | *owner behavioral happening* Universal Behavioral Happening |
| predecessor behavioral change part | *group-step* |
| successor behavioral change part | *group-step* |
| successor behavioral change part | *finish/abort* |
| successor | *previous succession* start-abort |

### 6.4.2.27 Instance: becomes false

**Class:** Fact Change

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| ownedType | *package* Happening & Change Library |

### 6.4.2.28 Instance: becomes true

**Class:** Fact Change

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| ownedType | *package* Happening & Change Library |

### 6.4.2.29 Instance: End

**Class:** Behavioral Change

**Description**

**End** is a **Behavioral Change** that manifests the end of a **Behavioral Happening**. The end can occur because of

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* endPart |
| general | *Generalization* |
| general | *Generalization* |
| general | *Generalization* |
| ownedType | *package* Happening & Change Library |

### 6.4.2.30 Instance: endPart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* End |
| owned behavioral change part | *owner behavioral happening* Universal Behavioral Happening |
| successor | *previous succession* startseq-end |
| successor | *previous succession* interationend-end |
| successor | *previous succession* cancel-end |
| successor | *previous succession* compensate-end |
| successor | *previous succession* start-end |

### 6.4.2.31 Instance: Error

**Class:** Behavioral Change

#### Description

**Error** is a **Behavioral Change** that manifests that an error has occurred that will lead to the **End** of the **Behavioral Happening**. The source of the **Error** is always internal to the **Behavioral Happening**.

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* errorPart |
| induced behavioral change | *behavioral change context* Error Process |
| ownedType | *package* Happening & Change Library |
| specific | *generalization* Generalization |

### 6.4.2.32 Instance: errorPart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Error |

| | |
|---|---|
| owned behavioral change part | *owner behavioral happening* Universal Behavioral Happening |
| predecessor behavioral change part | *error handling* |
| successor | *previous succession* start-error |

### 6.4.2.33 Instance: Finish

**Class:** Behavioral Change

#### Description

**Finish** is a **Behavioral Change** that manifests the normal **End** of a **Behavioral Happening**

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* finishPart |
| ownedType | *package* Happening & Change Library |
| specific | *generalization* Generalization |

### 6.4.2.34 Instance: finishPart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Finish |
| owned behavioral change part | *owner behavioral happening* Universal Behavioral Happening |
| predecessor behavioral change part | *start/start* |
| predecessor behavioral change part | *finish/abort* |
| successor | *previous succession* start-finish |

### 6.4.2.35 Instance: Generalization

**Class:** Generalization

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| | general End |
| generalization | *specific* Finish |

### 6.4.2.36 Instance: Generalization

**Class:** Generalization

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| | general End |
| generalization | *specific* Error |

### 6.4.2.37 Instance: Generalization

**Class:** Generalization

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| | general End |
| generalization | *specific* Abort |

### 6.4.2.38 Instance: Happening & Change Library

**Class:** Package

**Description**

User (M1) library capturing commonly needed aspects of behavioral happenings as instances of the class in the **Happening & Change** model. The library defines:

- **Behavioral Changes** to represent various behavior lifecycle events, such as starting and ending of individual **Behavioral Happenings**.

- A **Behavioral Happenings** called the **Universal Behavioral Happening**. It is a generalization of all M1 dynamic models (see the **Composition Model**).
- **Behavioral Change Parts** of the **Universal Behavioral Happening** for the various behavioral changes, such as startPart and finishPart. These are typed by the various M1 changes, such as Start and End Behavioral Changes
- Successions between the **Behavioral Change Parts** above for universal constraints, such as the End being after the Start.

**Links**

| Played End | *Opposite End* |
|---|---|
| nestedPackage | *nestingPackage* BPDM Library |
| package | *ownedType* Universal Behavioral Happening |
| package | *ownedType* becomes true |
| package | *ownedType* becomes false |
| package | *ownedType* Error |
| package | *ownedType* Finish |
| package | *ownedType* Start |
| package | *ownedType* End |
| package | *ownedType* Abort |

### 6.4.2.39 Instance: start-abort

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startPart |
| owned succession | *owner course* Universal Behavioral Happening |
| previous succession | *successor* abortPart |

### 6.4.2.40 Instance: start-end

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startPart |
| owned succession | *owner course* Universal Behavioral Happening |
| previous succession | *successor* endPart |

### 6.4.2.41 Instance: start-error

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startPart |
| owned succession | *owner course* Universal Behavioral Happening |
| previous succession | *successor* errorPart |

### 6.4.2.42 Instance: start-finish

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startPart |
| owned succession | *owner course* Universal Behavioral Happening |
| previous succession | *successor* finishPart |

### 6.4.2.43 Instance: Start

**Class:** Behavioral Change

**Description**

**Start** is a **Behavioral Change** that manifests the start of a **Behavioral Happening**.

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* startPart |
| ownedType | *package* Happening & Change Library |

### 6.4.2.44 Instance: startPart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Start |
| owned behavioral change part | *owner behavioral happening* Universal Behavioral Happening |
| predecessor | *next succession* start-error |
| predecessor | *next succession* start-compensate |
| predecessor | *next succession* start-cancel |
| predecessor | *next succession* start-finish |
| predecessor | *next succession* start-end |
| predecessor | *next succession* start-iterationend |
| predecessor | *next succession* start-abort |
| successor behavioral change part | *start/start* |

### 6.4.2.45 Instance: Universal Behavioral Happening

**Class:** Behavioral Happening

#### Description

**Behavioral Happening** that produces common behavior lifecycle changes, such as **Start** or **End**.

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral step type | *behavioral happening usage* Step Group |

| | |
|---|---|
| behavioral step type | *behavioral happening usage* Enclosed Step |
| behavioral step type | *behavioral happening usage* Racing Contestant |
| behavioral step type | *behavioral happening usage* Activity 1 |
| general | *Generalization* |
| general | *Generalization* |
| ownedType | *package* Happening & Change Library |
| owner behavioral happening | *owned behavioral change part* abortPart |
| owner behavioral happening | *owned behavioral change part* endPart |
| owner behavioral happening | *owned behavioral change part* finishPart |
| owner behavioral happening | *owned behavioral change part* errorPart |
| owner behavioral happening | *owned behavioral change part* startPart |
| owner course | *owned succession* start-end |
| owner course | *owned succession* start-abort |
| owner course | *owned succession* start-finish |
| owner course | *owned succession* start-error |

## 6.5 Processing Behavior Model

### 6.5.1 Introduction

The Processing Behavior Model enables behavioral happenings to be ordered in time as parts of other behavioral happenings (see the Happening and Change model). Vendors and users can define their own execution patterns with connections between these behavioral parts. The model predefines a specific connection for races, where behavioral happenings start at the same time and abort each other when the first finishes, and for part groups that abort the steps inside them. It also defines a change condition for detecting lifecycle events in behavioral happenings. The Processing Behavior Model is the most specialized model in BPDM that still covers all of orchestration and choreography (see the Activity Model and Interaction Protocol Model).

The Processing Behavior Model introduces:

- Courses with parts that behavioral happenings can play (Processing Behavior and Behavioral Steps). The course orders the happenings in time according to their behavioral changes, such as when they start and end (Processing Successions).
- Connections for behavioral steps that establish execution rules for connected steps (Compound Behavioral Connection). One of these is a connection between steps that all start at the same time, and where the first one to finish aborts the others (Race Connection and Race Behavior). Another connects groups that can abort their enclosed steps (Group Abort Connection and Group Abort Behavior).

- Behavioral steps for subprocessing behaviors (Processing Steps). These enable reuse of the same processing behavior.
- Behavioral steps for detecting changes in conditions, such as changes in time, facts, or behavior (Change Condition Step). For example, a change condition step can detect the passing of a certain point in time, a change in the truth of a statement due to changes in facts, and the completion of a happening, such as the arrival of a message.
- Groups of behavioral steps (Behavioral Step Group), where the group has its own behavioral change parts, such as for starting and ending.

Behavioral Steps are Typed Course Parts where the type is a Behavioral Happening. They are owned by Courses called Processing Behaviors. This enables processing behaviors to order happenings in time, as in the steps of a process model and or the interactions in a choreography. For example, the steps in a selling process are behavioral steps played by behavioral happenings such as packing and shipping. Individual selling processes (M0 performances, enactments, or executions of selling) can have a behavioral step played by an individual packing happening and another behavioral step played by an individual shipping happening.

Behavioral Steps are of two kinds:

- Processing Steps have Processing Behaviors as types, enabling them to "invoke" other behavioral happenings. The example selling process above is a processing behavior, where packing and shipping are the happening types of its processing steps.
- Change Condition Steps are behavioral steps that detect changes in Conditions, including time, facts (see the Happening and Change Model), or behavior. Change condition steps in user models (M1) are always typed by the Universal Behavioral Happening or subtypes of it that have no behavioral steps.

Processing Successions are Successions (see the Course Model) that can order the behavioral change parts of the steps, such as the start and end parts of packing or shipping. In the selling process example, a processing succession might have the packing part as source and the end part as internal source, while the shipping part is the target, and the start part is the internal target. This means packing must end before shipping starts. Processing Successions do not need to have behavioral steps on their ends, they can have untyped course parts also, such as gateways in BPMN, but they must have something on each end, as all successions do. For convenience, processing successions that do not specify internal source or target parts will have the same effect as processing successions where these are the end parts and start parts, respectively.

Immediate Processing Successions are Processing Successions that are also Immediate Successions (see the Course Model), for specifying changes playing behavioral steps happen at the same time. This is used in Process Behavior's user (M1) library, see Compound Behavioral Connections below.

Processing Successions can order change condition steps also. For example, a process can perform one step, then perform a time change condition step to wait for a certain duration to elapse, then another step. This is enabled by change condition steps at M1 being typed by the Universal Behavioral Happening (see the Happening and Change Model), to define the standard behavioral change parts, for example startPart and endPart.

Connected Part Bindings are Elements specifying that individuals playing the part at an end of a connection also play a part within the connection. For example, one of the interactions between businesses in a choreography might be a subchoreography composed of many communications between the businesses. Businesses playing a particular role in the larger choreography also play one of the roles in the subchoreography. Bindable Connections are defined just to categorize those connections that can carry part bindings. The player is part of the composite owning the bindable connection. The played is part of the bindable connection. The binding requires the (M0) individuals playing these parts to be the same. They are found by navigating from an individual composite, to the player individuals, and to the played individuals in the connection part of the same composite. The two sets of individuals found this way must be exactly the same. Connected part bindings are different from connections because part bindings are about which individuals are playing certain parts in a whole, whereas connections are about links between the individuals themselves due to playing parts in the whole. As a convenience, it is assumed that a connection typed by a composite that has only one (non-connection) part implies bindings where that one part is played by all the parts at all the ends of the connector. This is useful for symmetrical connectors (see Race Connector below for an application).

Compound Behavioral Connections are Connections between Behavioral Steps that are also Typed Parts, enabling connections to reuse the same composite for connecting steps. BPDM defines two kinds of compound behavioral connections:

- Race Connections are Compound Behavioral Connections that are always typed by Race Behavior, an M1 instance of Processing Behavior defined in the Processing Behavior user (M1) library. Race Behavior ensures that all the behavioral steps connected by Race Connection start at the same time, and that the first one to finish aborts the others (see the Happening and Change Model)[10] Race Behavior contains:
- One step, called the Contestant, which is bound to all the steps connected by the M1 race connection (see Connected Part Binding above). This ensures that all the contestants are treated the same way.
- Two immediate processing successions connecting the Contestant to itself. One succession refers to the start part of the Contestant on both ends (see the Happening and Change Model), specifying that all the contestant behavioral happenings start at the same time. The other succession has the finish part on one end and the abort part on the other, specifying that any contestant happening that finishes will be accompanied by a simultaneous abort of the others. This succession has the Irreflexive condition applied (see the Composition Model), to prevent the finishing contestant from aborting itself.

When a race connection is created between behavioral steps, it implies part bindings between the connected steps and the Contestant in Race Behavior, with Contestant on the played end (see Connected Part Binding above). The part bindings ensure that any individual M0 happening playing the connected steps will also play the Contestant, establishing the start-start and finish-abort successions between the connected steps, and the temporal constraints on the individual happenings. The Race Behavior above can be the type for any connector that is also a typed part, but Race Connection is always typed by Race Behavior, for convenience.

- Group Abort Connections are Compound Behavioral Connections that are always typed by Group Abort Behavior, an M1 instance of Processing Behavior defined in the Processing Behavior user (M1) library. It is applied to behavioral step groups and their enclosed steps to ensure that the steps are aborted when the group is. Group Abort Behavior contains:
- Two steps, one for the group and one for its enclosed steps (Step Group and Enclosed Step). The first is bound to an M1 behavioral step group and the second to each step in the group (see Connected Part Binding above).
- One immediate processing succession between the two steps above. The source is Step Group and the target is Enclosed Step. It refers to the abort part on both ends (see the Happening and Change Model), specifying that any group behavioral happening that aborts will be accompanied by a simultaneous abort of the enclosed step happenings.

When a group abort connection is created between a behavior step group and its steps, it implies a part binding between Step Group in the Group Abort Behavior and the connected group, with Step Group on the played end (see Connected Part Binding above). Similarly, it implies bindings between Enclosed Step and the steps in the group. The part bindings ensure that any individual M0 happening playing the connected group will also play the Step Group, and any individual playing the connected steps will also play the Enclosed Step, establishing the abort-abort successions between the connected group and steps, and the temporal constraints on the individual happenings. The Group Abort Behavior above can be the type for any connector that is also a typed part, but Group Abort Connection is always typed by Group Abort Behavior, for convenience.

10. See the BPMN package for application to event-based gateways and intermediate events attached to activities.

Users and vendors can capture their own execution patterns by defining M1 processing behaviors to use as the type of compound behavioral connections. For example, some vendors might have an option on races to not abort the losing processes. This is a variation on the Race Behavior that does not have the finish-abort successions. It can be defined as an M1 instance of Compound Behavioral Connection that is always typed by the vendor-defined variant Race Behavior.

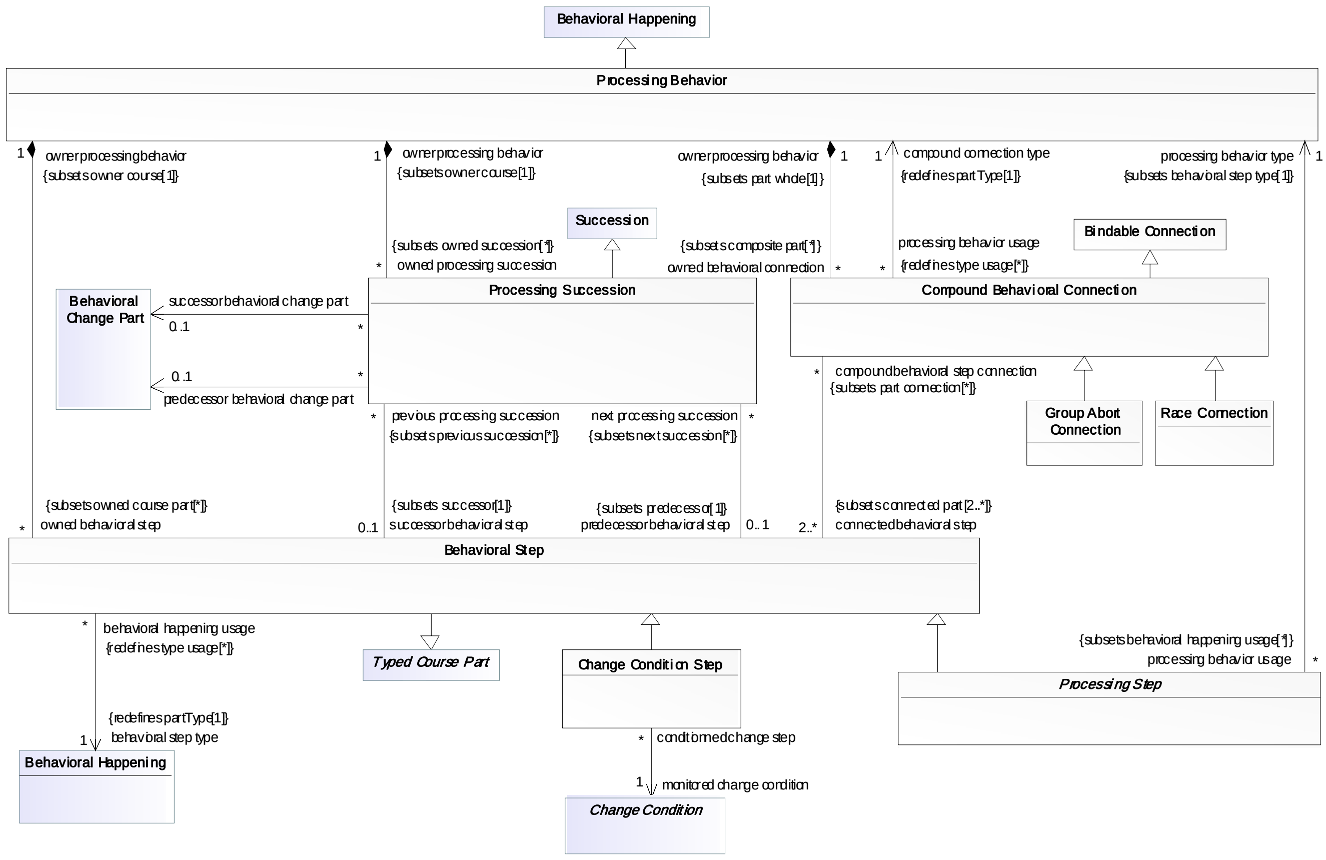
Behavioral Change Conditions are Change Conditions for detecting behavioral changes in happenings, for example the start and ending of a happening. It specifies the happening with a behavioral step, such as a step in a process or interaction in a choreography, and specifies the change with a change part, such as the parts for starting and ending (see the Happening and Change Model). A behavioral change condition can be the condition for a change condition step, enabling detection of the starting and ending of happenings identified by behavioral steps. For example, a behavioral change condition can refer to a message part and the finish part in it to specify that the message has arrived (BPDM represents messages as processes in themselves, see Simple Interaction Model).

Behavioral Step Groups are Part Groups (see the Composition Model) that enclose Behavioral Steps, and are also Behavioral Steps themselves, typed by the Universal Behavioral Happening in user models (M1). This gives a group of behavioral steps as a whole the capacity to produce start and end changes playing the standard behavioral change parts, such as startPart and endPart. For example, most process languages have a way of modeling subprocesses without defining a separate process. This is a behavioral step group.

### 6.5.2 Metamodel Specification

The Processing Behavior Model enables behavioral happenings to be ordered in time as parts of other behavioral happenings (see the Happening and Change model). Vendors and users can define their own execution patterns with connections between these behavioral parts. The model predefines a specific connection for races, where behavioral happenings start at the same time and abort each other when the first finishes. It also defines a change condition for detecting lifecycle events in behavioral happenings. The Processing Behavior Model is the most specialized model in BPDM that still covers all of processes and interactions (orchestration and choreography, see the Activity and Interaction Protocol Models).

### 6.5.2.1 Processing Behavior



Behavioral Happening
Processing Behavior
owner processing behavior
{subsets owner course[1]}
owner processing behavior
{subsets owner course[1]}
owner processing behavior
{subsets part whole[1]}
compound connection type
{redefines partType[1]}
processing behavior type
{subsets behavioral step type[1]}
Succession
{subsets owned succession[*]}
owned processing succession
{subsets composite part[*]}
owned behavioral connection
processing behavior usage
{redefines type usage[*]}
Bindable Connection
Behavioral Change Part
successor behavioral change part
predecessor behavioral change part
Processing Succession
Compound Behavioral Connection
compound behavioral step connection
{subsets part connection[*]}
Group Abort Connection
Race Connection
previous processing succession
{subsets previous succession[*]}
next processing succession
{subsets next succession[*]}
{subsets owned course part[*]}
owned behavioral step
{subsets successor[1]}
successor behavioral step
{subsets predecessor[1]}
predecessor behavioral step
{subsets connected part[2..*]}
connected behavioral step
Behavioral Step
behavioral happening usage
{redefines type usage[*]}
Typed Course Part
Change Condition Step
{subsets behavioral happening usage[*]}
processing behavior usage
Processing Step
{redefines partType[1]}
behavioral step type
Behavioral Happening
conditioned change step
monitored change condition
Change Condition

**Figure 6.39 - Processing Behavior**

### 6.5.2.2 Connected Part Binding

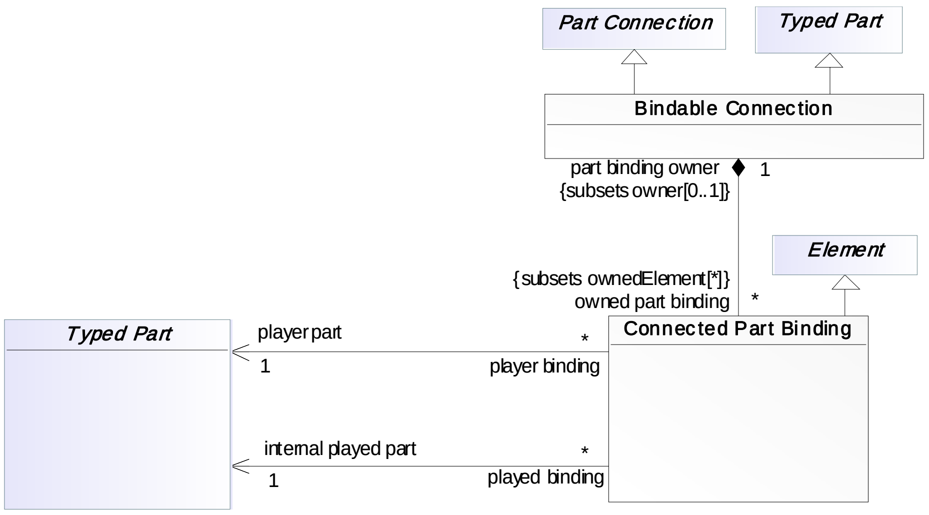


**Figure 6.40 - Connected Part Binding**

### 6.5.2.3 Immediate Process Succession

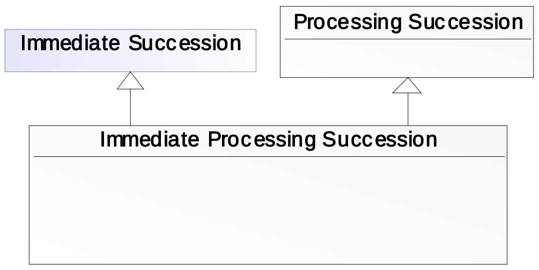


**Figure 6.41 - Immediate Process Succession**

### 6.5.2.4 Process Behavior Library: 'Racing' Processing Behavior instance

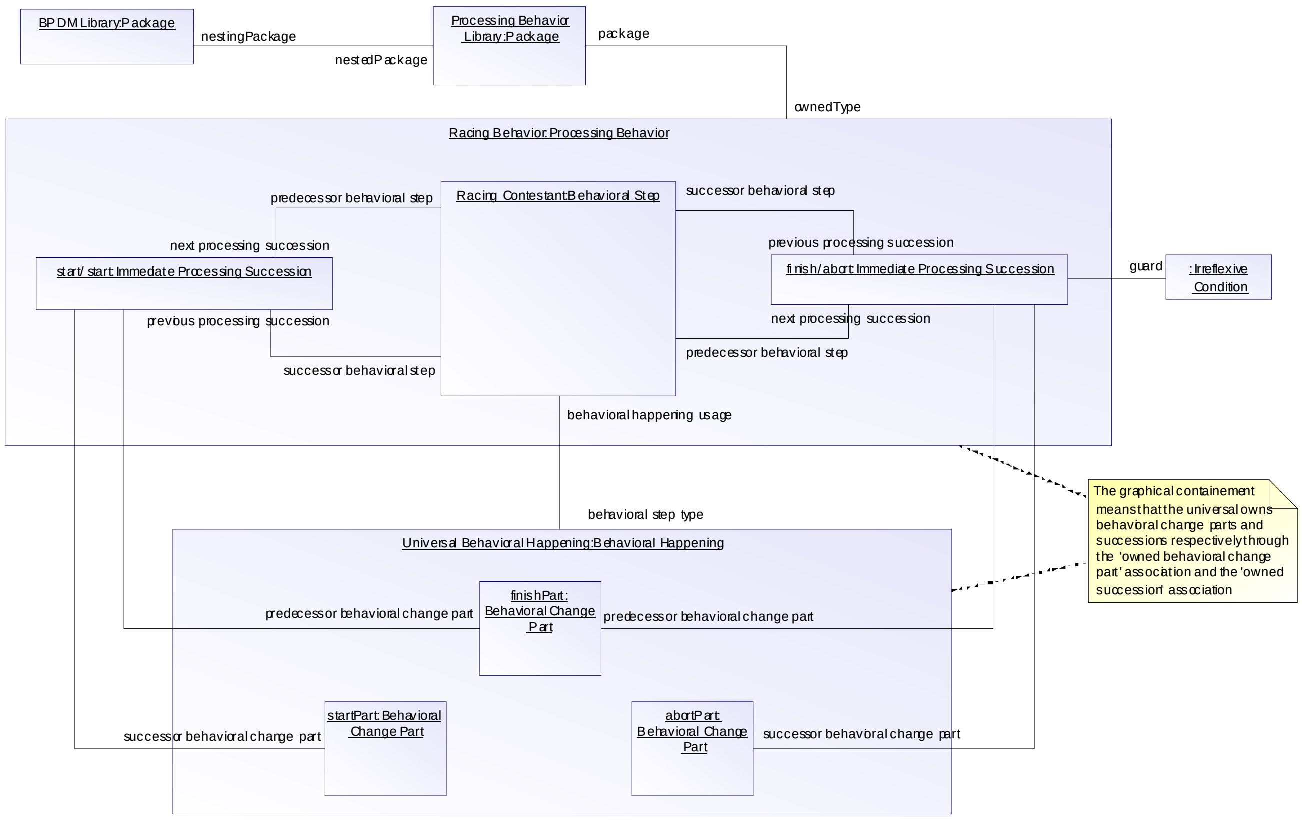

BPDM Library:Package
nestingPackage
nestedPackage
Processing Behavior Library:Package
package
ownedType
Racing Behavior:Processing Behavior
predecessor behavioral step
Racing Contestant:Behavioral Step
successor behavioral step
next processing succession
start/ start: Immediate Processing Succession
previous processing succession
successor behavioral step
previous processing succession
finish/abort: Immediate Processing Succession
next processing succession
predecessor behavioral step
guard
: Irreflexive Condition
behavioral happening usage
behavioral step type
The graphical containement means that the universal owns behavioral change parts and successions respectively through the 'owned behavioral change part' association and the 'owned succession' association
Universal Behavioral Happening:Behavioral Happening
predecessor behavioral change part
finishPart: Behavioral Change Part
predecessor behavioral change part
successor behavioral change part
startPart: Behavioral Change Part
abortPart: Behavioral Change Part
successor behavioral change part

**Figure 6.42 - Process Behavior Library: 'Racing' Processing Behavior instance**

### 6.5.2.5 Processing Behavior Library: 'Group Abort Behavior'

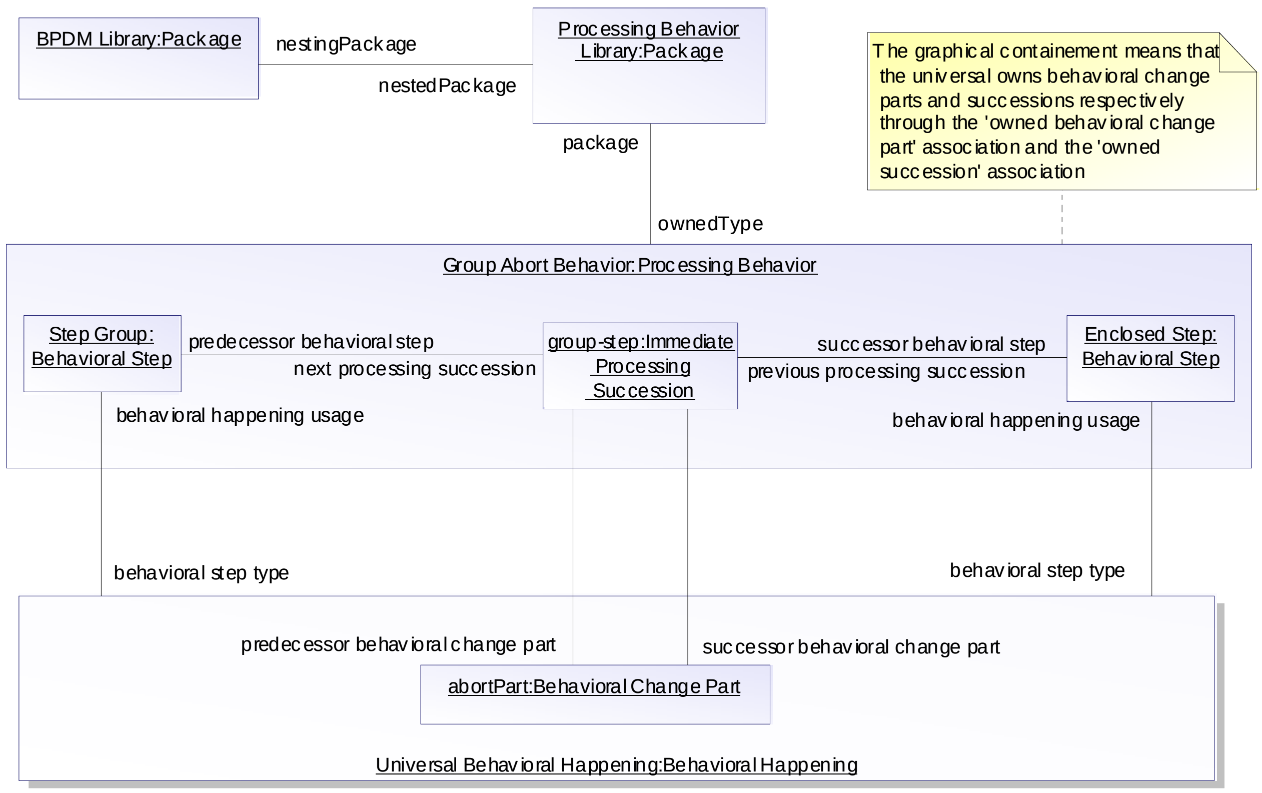

BPDM Library:Package
nestingPackage
nestedPackage
Processing Behavior Library:Package
package
ownedType
The graphical containement means that the universal owns behavioral change parts and successions respectively through the 'owned behavioral change part' association and the 'owned succession' association
Group Abort Behavior:Processing Behavior
Step Group: Behavioral Step
predecessor behavioral step
next processing succession
group-step:Immediate Processing Succession
successor behavioral step
previous processing succession
Enclosed Step: Behavioral Step
behavioral happening usage
behavioral happening usage
behavioral step type
behavioral step type
predecessor behavioral change part
successor behavioral change part
abortPart:Behavioral Change Part
Universal Behavioral Happening:Behavioral Happening

**Figure 6.43 - Processing Behavior Library: 'Group Abort Behavior'**

### 6.5.2.6 Behavioral Change Condition

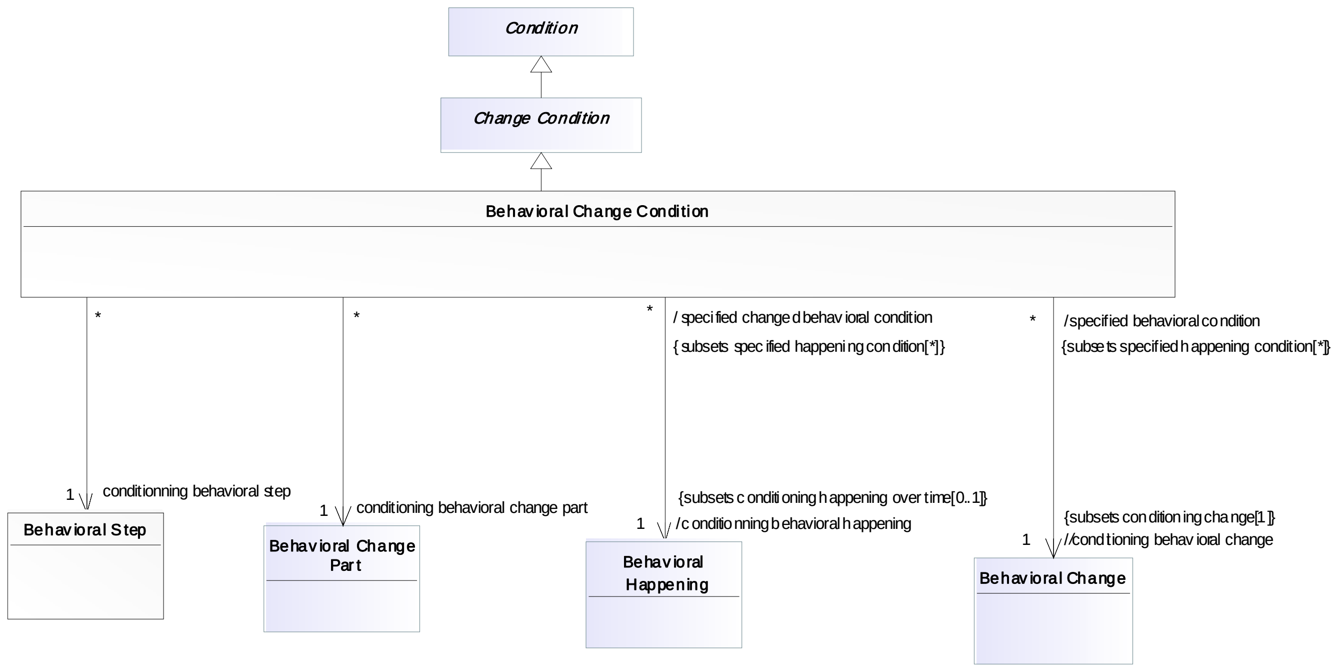

Condition
Change Condition
Behavioral Change Condition
*
*
*
/ specified changed behavioral condition
{ subsets specified happening condition[*]}
*
/specified behavioral condition
{subsets specified happening condition[*]}
1
conditionning behavioral step
Behavioral Step
1
conditioning behavioral change part
Behavioral Change Part
{subsets conditioning happening over time[0..1]}
1
/conditionning behavioral happening
Behavioral Happening
{subsets conditioning change[1]}
1
//conditioning behavioral change
Behavioral Change

Figure 6.44 - Behavioral Change Condition

### 6.5.2.7 Behavioral Step Group

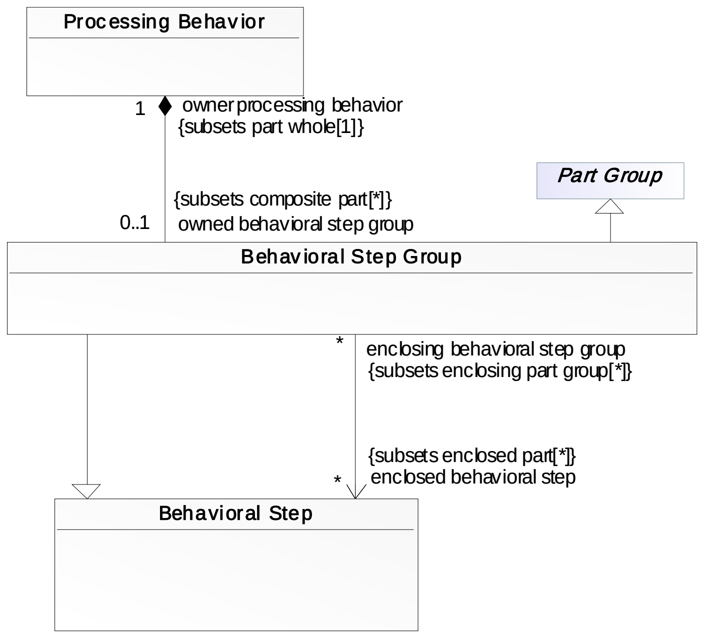

Processing Behavior
1
owner processing behavior
{subsets part whole[1]}
Part Group
{subsets composite part[*]}
0..1
owned behavioral step group
Behavioral Step Group
*
enclosing behavioral step group
{subsets enclosing part group[*]}
{subsets enclosed part[*]}
*
enclosed behavioral step
Behavioral Step

Figure 6.45 - Behavioral Step Group

### 6.5.2.8 Behavioral Change Condition

**Namespace:** Processing Behavior
**isAbstract:** No
**Generalization:** “Change Condition”

#### Description

A **Behavioral Change Condition** is a kind of **Change Condition** for detecting a **Behavioral Change** in a particular **Behavioral Happening**, for example the start and ending of this **Behavioral Happening**.

that is based on the occurrence of a **Behavioral Change** in a particular **Behavioral Happening** as defined by the **conditionning behavioral step**.

Behavioral Change Conditions are Change Conditions for detecting behavioral changes in happenings, for example the start and ending of a happening.

It specifies the happening with a behavioral step, such as a step in a process or interaction in choreography, and specifies the change with a change part, such as the parts for starting and ending (see the Happening and Change Model). A behavioral change condition can be the condition for a change condition step, enabling detection of the starting and ending of happenings identified by behavioral steps. For example, a behavioral change condition can refer to a message part and the finish part in it to specify that the message has arrived (BPDM represents messages as processes in themselves, see Simple Interaction Model).

#### Associations

| | |
|---|---|
| conditioning behavioral change part : Behavioral Change Part [1] | Behavioral Change Part that specifies the Behavioral Change that is the source of the condition, such as the start (startPart) or end (endPart) |

| | |
|---|---|
| conditioning behavioral change : Behavioral Change [1] | Behavioral Change that specifies the Behavioral Change Condition. This is derived from the Behavioral Change Part that defines the Behavioral Change Condition<br>Subsets *conditioning change* |

| | |
|---|---|
| conditionning behavioral happening : Behavioral Happening [1] | Behavioral Happening that specifies the context of the Behavioral Change that defines the condition. This is derived from conditionning behavioral step of the condition.<br>Subsets *conditioning happening over time* |

| | |
|---|---|
| conditionning behavioral step : Behavioral Step [1] | Behavioral Step that is the source of the condition, such as an activity in a process or an interaction in a protocol |

#### Constraint

[1] The **conditioning behavioral change part** must be a **Behavioral Change Part** of the type of the **conditionning behavioral step**

self.**conditioning behavioral change part** in self.**conditionning behavioral step ->behavioral step type ->owned behavioral change part**

### 6.5.2.9 Behavioral Step

**Namespace:** Processing Behavior
**isAbstract:** No
**Generalization:** “Typed Course Part”

#### Description

A **Behavioral Step** is a kind of **Typed Course Part** where the type is a **Behavioral Happening**. This enables the ordering over time of **Behavioral Happenings** in the context of a **Processing Behavior** . As such, **Behavioral Steps** can be connected by **Processing Successions**.

#### Associations

| | |
|---|---|
| behavioral step type : Behavioral Happening [1] | Behavioral Happening typing the Behavioral Step.<br>The default behavioral step type is the Universal Behavioral Happening<br>Redefines *partType* |

| | |
|---|---|
| compound behavioral step connection : Compound Behavioral Connection [*] | Compound Behavioral Connection indicating that the lifecycle of the Behavioral Step is tied to the life cycle of other Behavioral Steps.<br>Subsets *part connection* |

| | |
|---|---|
| next processing succession : Processing Succession [*] | Processing Succession that enables the Behavioral Step as its predecessor behavioral step<br>Subsets *next succession* |

| | |
|---|---|
| previous processing succession : Processing Succession [*] | Processing Succession that enables the Behavioral Step as its successor behavioral step<br>Subsets *previous succession* |

### 6.5.2.10 Behavioral Step Group

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Behavioral Step” “Part Group”

#### Description

A **Behavioral Step Group** is a kind of **Part Group** that is also a **Behavioral Step** typed by the **Universal Behavioral Happening** in user models (M1). This gives a group of **Behavioral Steps** as a whole the capacity to produce start and end changes playing the standard behavioral change parts, such as **startPart** and **endPart**.

For example, most process languages have a way of modeling sub-processes without defining a separate process. This is a **Behavioral Step Group**.

### Associations

| | |
|---|---|
| enclosed behavioral step : Behavioral Step [*] | Behavioral Step being part of the Behavioral Step Group<br>Subsets *enclosed part* |

### 6.5.2.11 Bindable Connection

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Part Connection” “Typed Part”

#### Description

A **Bindable Connection** is a kind of **Part Connection** defined just to categorize those connections that can carry **Connected Part Binding**.

#### Associations

| | |
|---|---|
| owned part binding : Connected Part Binding [*] | Connected Part Binding owned by the Composite<br>Subsets *ownedElement* |

### 6.5.2.12 Change Condition Step

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Behavioral Step”

#### Description

A  **Change Condition Step** is a kind of **Typed Course Part** that monitors the occurrence of a **Change Condition** and that has an effect on the course of a **Processing Behavior**. For instance, a **Change Condition Step** can be used to react to the **Abort** of a specific **Behavioral Happening**.

#### Associations

| | |
|---|---|
| monitored change condition : Change Condition [1] | Change Condition being monitored |

#### BPMN Notation

**Change Condition Step** shape with the marker of the **Compensate** instance of **Behavioral Change**

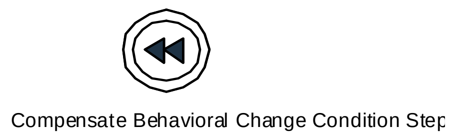


**Figure 6.46 - Change Condition Step monitoring a 'Compensate' Behavioral Change Condition**

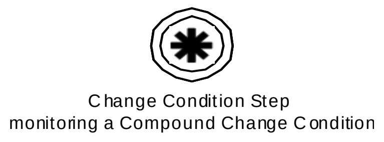


Figure 6.47 - Change Condition Step monitoring a Compound Change Condition

**Change Condition Step** shape with a **Fact Change** as a maker

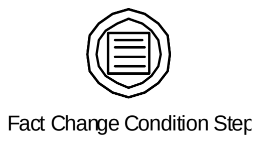


Figure 6.48 - Change Condition Step monitoring a Fact Change Condition

**Change Condition Step** shape with a **Time Change** as a maker

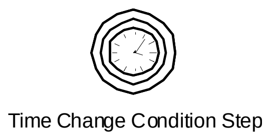


Figure 6.49 - Change Condition Step monitoring a Time Change Condition

This symbol is a a circle, with an open center. The circle MUST be drawn with a double thin black line. It can alternatively represent:

1. **Behavioral Change Parts** that are not typed by **Start** or **End**
2. **Change Condition Steps**

Markers can be placed within the circle to indicate the nature of the **Change** associated with the **Behavioral Change Part** or **Change Condition Step**

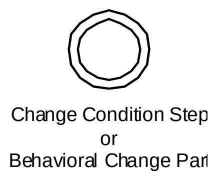


Figure 6.50 - Change Condition Step Notation or Behavioral Change Part

### 6.5.2.13 Compound Behavioral Connection

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Bindable Connection”

#### Description

A **Compound Behavioral Connection** is a **Part Connection** that enables dedicated lifecycle rule connections to apply between **Behavioral Steps**. These rules are described by the **compound connection type** of the **Compound Behavioral Connection**, which is itself a **Processing Behavior**.

This makes **Compound Behavioral Connection** be itself a **Typed Part**.

#### Associations

| | |
|---|---|
| compound connection type : Processing Behavior [1] | Processing Behavior typing the Compound Behavioral Connection and specifying the lifecycle rules (start/start, abort/abort) tying all Behavioral Steps connected by the Compound Behavioral Connection<br>Redefines *partType* |

| | |
|---|---|
| connected behavioral step : Behavioral Step [2..*] | Behavioral Step connected by the Compound Behavioral Connection<br>Subsets *connected part* |

### 6.5.2.14 Connected Part Binding

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** "Element"

#### Description

A **Connected Part Binding** is an **Element** specifying that individuals playing the part at an end of a **Part Connection** also play a **Part** within the connection. For example, one of the interactions between businesses in a choreography might be a subchoreography composed of many communications between the businesses. Businesses playing a particular role in the larger choreography also play one of the roles in the subchoreography.

The player is part of the composite owning the bindable connection. The played is part of the bindable connection. The binding requires the (M0) individuals playing these parts to be the same. They are found by navigating from an individual composite, to the player individuals, and to the played individuals in the connection part of the same composite. The two sets of individuals found this way must be exactly the same.

**Connected Part Binding** is different from **Part Connection** because part bindings are about which individuals are playing certain parts in a whole, whereas connections are about links between the individuals themselves due to playing parts in the whole. As a convenience, it is assumed that a connection typed by a composite that has only one (non-connection) part implies bindings where that one part is played by all the parts at all the ends of the connector. This is useful for symmetrical connectors.

#### Associations

| | |
|---|---|
| internal played part : Typed Part [1] | the played is part of the bindable connection |

| | |
|---|---|
| player part : Typed Part [1] | The player is part of the composite owning the bindable connection |

### 6.5.2.15 Group Abort Connection

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Compound Behavioral Connection”

#### Description

A **Group Abort Connection** is a kind of **Compound Behavioral Connection** which has for **compound connection type** the **Group Abort Behavior**, an M1 instance of **Processing Behavior** defined in the **Processing Behavior Library**user (M1) library.

It is applied to **Behavioral Step Groups** and their enclosed steps to ensure that the steps are aborted when the group is. (See more details in **Group Abort Behavior**).

### 6.5.2.16 Immediate Processing Succession

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Immediate Succession” “Processing Succession”

#### Description

An **Immediate Processing Succession** is a **Processing Succession** that is immediate, as defined by the **Immediate Succession**

### 6.5.2.17 Processing Behavior

**Namespace:** Processing Behavior
**isAbstract:** No
**Generalization:** “Behavioral Happening”

#### Description

A **Processing Behavior** is of **Behavioral Happening**s that order happenings in time, as in the activities of a process model and or the interactions in a choreography.

**Processing Behavior** introduces capabilites shared by both choreography and orchestration:

- Its steps are typed by **Behavioral Happening**s which provide them with start/end capabilities
- As a **Course** it can organize its part with **Succession**. It adds the ability to order its steps according to their start and ends (**Processing Succession**).
- Rich connections can be establish between its steps to enable time sychronization between them (**Compound Behavioral Connection**).
- The reuse of the same **Processing Behavior** enable .is enabled by (**Processing Step**).
- Detection of changes in conditions, such as changes in time, facts, or behavior can be to influence its course (**Change Condition Step**)
- Its steps can be organize in groups to which start/end constraints can be applied (**Behavioral Step Group**)

### Associations

| | |
|---|---|
| owned behavioral connection : Compound Behavioral Connection [*] | Compound Behavioral Connection owned by the Processing Behavior<br>Subsets *composite part* |

| | |
|---|---|
| owned behavioral step group : Behavioral Step Group [0..1] | Behavioral Step Group owned by the Processing Behavior<br>Subsets *composite part* |

| | |
|---|---|
| owned behavioral step : Behavioral Step [*] | Behavioral Step owned by the Processing Behavior<br>Subsets *owned course part* |

| | |
|---|---|
| owned processing succession : Processing Succession [*] | Processing Succession owned by the Processing Behavior<br>Subsets *owned succession* |

## 6.5.2.18 Processing Step

**Namespace:** Processing Behavior
**isAbstract:** Yes
**Generalization:** “Behavioral Step”

### Description

**Processing Steps** is a kind of **Behavioral Step** which type is a **Processing Behavior**. This enables it to “invoke” other **Processing Behavior** and to build **Processing Behavior** composites (made of sub- **Processing Behaviors**).

### Associations

| | |
|---|---|
| processing behavior type : Processing Behavior [1] | specifies the type of the Processing Step.<br>The default processing behavior type is the Universal Behavioral Happening<br>Subsets *behavioral step type* |

## 6.5.2.19 Processing Succession

**Namespace:** Processing Behavior
**isAbstract:**
**Generalization:** “Succession”

### Description

A **Processing Succession** is a kind of **Succession** that can order the **Behavioral Change Parts** of its **Behavioral Step** such as their start and end parts.

**Processing Succession** allows any combination of **Behavioral Change Part** to be connected.

End -> Start

Start -> Start

Start -> Abort

etc.

A **Processing Succession** doesn't need to have **Behavioral Steps** on its ends, it can have untyped course parts also, such as **Course Control Part**, but it must have something on each end, as all **Successions** do.

For convenience, a **Processing Succession** that do not specify **predecessor behavioral change part** or **successor behavioral change part** will have the same effect as a **Processing Succession** where these are respectively the end part and start part.

### Associations

| | |
|---|---|
| predecessor behavioral change part : Behavioral Change Part [0..1] | Behavioral Change Part of the predecessor behavioral step that is connected through the Processing Succession |

| | |
|---|---|
| predecessor behavioral step : Behavioral Step [0..1] | Behavioral Step that comes before another Course Part in a Processing Succession<br>Subsets *predecessor* |

| | |
|---|---|
| successor behavioral change part : Behavioral Change Part [0..1] | Behavioral Change Part of the successor behavioral step that is connected through the Processing Succession. |

| | |
|---|---|
| successor behavioral step : Behavioral Step [0..1] | Behavioral Step that comes after another Course Part in a Processing Succession<br>Subsets *successor* |

### Constraint

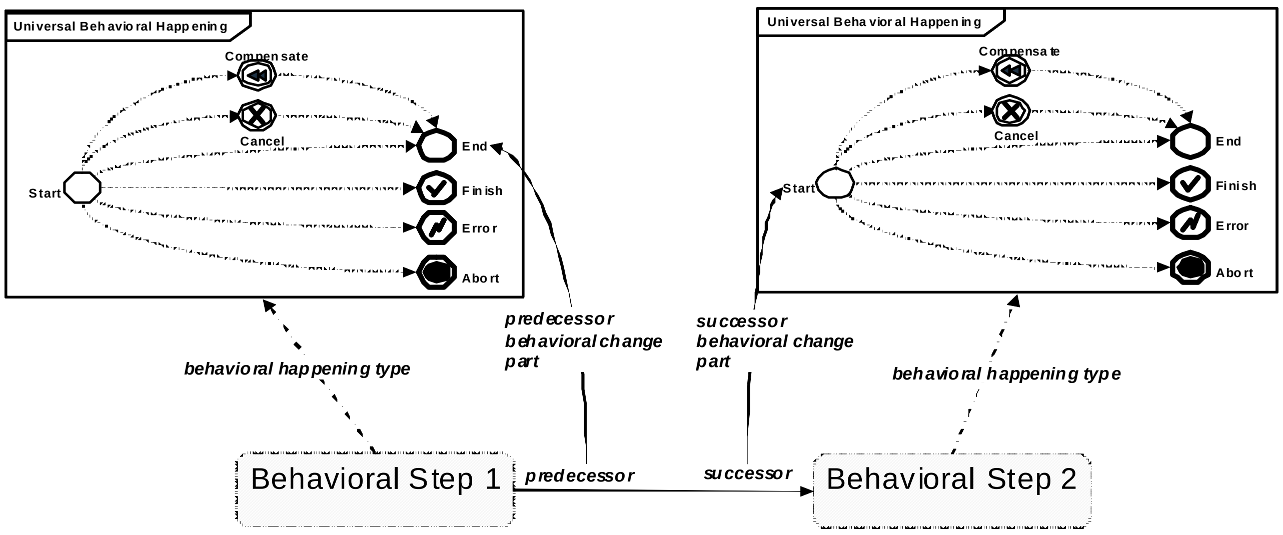
[2] The **predecessor behavioral change part** must be one of the **Behavioral ChanBehavioral Change**s of the **Behavioral Happening** that is the type of the **predecessor behavioral step**

self.**predecessor behavioral change part** in self.**predecessor behavioral step ->behavioral step type ->owned behavioral change part**

[2] The **successor behavioral change part** must be one of the **Behavioral Change**s of the **Behavioral Happening** that is the type of the **successor behavioral step**

self.**successor behavioral change part** in self.**successor behavioral step ->behavioral step type ->owned behavioral change part**

**Processing Succession & Behavioral Change Part**



Universal Behavioral Happening
Compensate
Cancel
Start
End
Finish
Error
Abort
Universal Behavioral Happening
Compensate
Cancel
Start
End
Finish
Error
Abort
behavioral happening type
predecessor behavioral change part
successor behavioral change part
behavioral happening type
Behavioral Step 1
predecessor
successor
Behavioral Step 2

**Figure 6.51 - Processing Succession & Behavioral Change Part**

### BPMN Notation

A Succession is line with a solid arrowhead that MUST be drawn with a solid single line

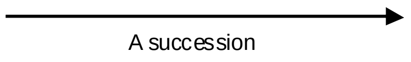


**Figure 6.52 - Succession Notation**

### 6.5.2.20 Race Connection

**Namespace:** Processing Behavior

**isAbstract:**

**Generalization:** “Compound Behavioral Connection”

#### Description

A **Race Connection** is a kind of **Compound Behavioral Connection** which has for **compound connection type** the **Racing Behavior**.

The **Racing Behavior** ensures that all the connected **Behavioral Step** start at the same time, and that the first one to finish aborts the others.

### 6.5.2.21 Instance: Enclosed Step

**Class:** Behavioral Step

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral happening usage | *behavioral step type* Universal Behavioral Happening |
| owned behavioral step | *owner processing behavior* Group Abort Behavior |
| successor behavioral step | *previous processing succession* group-step |

### 6.5.2.22 Instance: finish/abort

**Class:** Immediate Processing Succession

#### Description

This succession has the finish part on one end and the abort part on the other, specifying that any contestant happening that finishes will be accompanied by a simultaneous abort of the others. This succession has the Irreflexive condition applied (see the Composition Model), to prevent the finishing contestant from aborting itself.

#### Links

| Played End | *Opposite End* |
|---|---|
|  | successor behavioral change part abortPart |
|  | predecessor behavioral change part finishPart |
|  | guard Irreflexive Condition |
| next processing succession | *predecessor behavioral step* Racing Contestant |
| owned succession | *owner course* Racing Behavior |
| previous processing succession | *successor behavioral step* Racing Contestant |

### 6.5.2.23 Instance: Group Abort Behavior

**Class:** Processing Behavior

#### Description

**Group Abort Behavior** contains:

- Two steps, one for the group and one for its enclosed steps (Step Group and Enclosed Step). The first is bound to an M1 behavioral step group and the second to each step in the group (see Connected Part Binding above).
- One immediate processing succession between the two steps above. The source is Step Group and the target is Enclosed Step. It refers to the abort part on both ends (see the Happening and Change Model), specifying that any group behavioral happening that aborts will be accompanied by a simultaneous abort of the enclosed step happenings.

When a group abort connection is created between a behavior step group and its steps, it implies a part binding between Step Group in the Group Abort Behavior and the connected group, with Step Group on the played end (see Connected Part Binding above). Similarly, it implies bindings between Enclosed Step and the steps in the group. The part bindings ensure that any individual M0 happening playing the connected group will also play the Step Group, and any individual playing the connected steps will also play the Enclosed Step, establishing the abort-abort successions between the connected group and steps, and the temporal constraints on the individual happenings. The Group Abort Behavior above can be the type for any connector that is also a typed part, but Group Abort Connection is always typed by Group Abort Behavior, for convenience.

**Links**

| Played End | *Opposite End* |
|---|---|
| ownedType | *package* Processing Behavior Library |
| owner processing behavior | *owned behavioral step* Step Group |
| owner processing behavior | *owned processing succession* group-step |
| owner processing behavior | *owned behavioral step* Enclosed Step |

### 6.5.2.24 Instance: group-step

**Class:** Immediate Processing Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
|  | predecessor behavioral change part abortPart |
|  | successor behavioral change part abortPart |
| next processing succession | *predecessor behavioral step* Step Group |
| owned processing succession | *owner processing behavior* Group Abort Behavior |
| previous processing succession | *successor behavioral step* Enclosed Step |

### 6.5.2.25 Instance: Processing Behavior Library

**Class:** Package

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| nestedPackage | *nestingPackage* BPDM Library |
| package | *ownedType* Racing Behavior |
| package | *ownedType* Group Abort Behavior |

### 6.5.2.26 Instance: Racing Behavior

**Class:** Processing Behavior

**Description**

**Racing Behavior** contains:

- One step, called the Contestant, which is bound to all the steps connected by the M1 race connection (see Connected Part Binding above). This ensures that all the contestants are treated the same way.
- Two immediate processing successions connecting the Contestant to itself. One succession refers to the start part of the Contestant on both ends (see the Happening and Change Model), specifying that all the contestant behavioral happenings start at the same time. The other succession has the finish part on one end and the abort part on the other, specifying that any contestant happening that finishes will be accompanied by a simultaneous abort of the others. This succession has the Irreflexive condition applied (see the Composition Model), to prevent the finishing contestant from aborting itself.

**Links**

| Played End | *Opposite End* |
|---|---|
| ownedType | *package* Processing Behavior Library |
| owner course | *owned succession* finish/abort |
| owner course | *owned succession* start/start |
| owner processing behavior | *owned behavioral step* Racing Contestant |

### 6.5.2.27 Instance: Racing Contestant

**Class:** Behavioral Step

**Description**

**Behavioral Step** of the **Racing Behavior** is bound to all the steps connected by the M1 race connection to ensures that all the contestants are treated the same way.

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral happening usage | *behavioral step type* Universal Behavioral Happening |
| owned behavioral step | *owner processing behavior* Racing Behavior |
| predecessor behavioral step | *next processing succession* finish/abort |
| predecessor behavioral step | *next processing succession* start/start |
| successor behavioral step | *previous processing succession* finish/abort |
| successor behavioral step | *previous processing succession* start/start |

### 6.5.2.28 Instance: start/start

**Class:** Immediate Processing Succession

**Description**

This succession refers to the start part of the **Racing Contestant** on both ends (see the Happening and Change Model introduction), specifying that all the contestant behavioral happenings start at the same time.

**Links**

| Played End | *Opposite End* |
|---|---|
|  | predecessor behavioral change part finishPart |
|  | successor behavioral change part startPart |
| next processing succession | *predecessor behavioral step* Racing Contestant |
| owned succession | *owner course* Racing Behavior |
| previous processing succession | *successor behavioral step* Racing Contestant |

### 6.5.2.29 Instance: Step Group

**Class:** Behavioral Step

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral happening usage | *behavioral step type* Universal Behavioral Happening |
| owned behavioral step | *owner processing behavior* Group Abort Behavior |
| predecessor behavioral step | *next processing succession* group-step |

## 6.6 Simple Interaction Model

### 6.6.1 Introduction

The Simple Interaction Model enables interactions to be treated like any other step in a processing behavior, ordered in time, with start and end events. The model is the basis for flows between process steps and between participants in a choreography (see the Activity Model and the Interaction Protocol Model). The Simple Interaction Model is the most specialized model in BPDM that still has elements in common between orchestration and choreography.

The Simple Interaction Model provides:

- Interactions that have no subinteractions (Simple Interactions).
- Types for flowing entities (transferred item types).
- Expressions for changing which entities are flowing (transformation expression).
- Parts that interact within a processing behavior (Interaction Roles).

Interactions are Behavioral Steps that are also Connections, enabling them to have start and end changes, and be ordered in time. This is used to define reusable protocols and specify the way a process interacts with its environment (see the Interaction Protocol Model and the Activity Model). Interactive Parts are defined just to categorize those Typed Parts that can be connected by Interactions. The types of interactive parts establish requirements for the interacting individuals, for example, that they have a minimum security clearance or market capitalization.

Simple Interactions are interactions in which something is "transferred" from individuals playing one interactive part to individuals playing another interactive part. For example, a document, phone number, or package may be transferred from one department to another in a company. The transferred items must conform to a Type specified by the simple interaction. Simple Interactions can have an expression to change the item that arrives at the target based on the item flowing from the source. For example, a transformation may retrieve the zip code from an address flowing from the source to deliver the zip code to the target.

Simple Interactions in user (M1) models are always typed by the Universal Behavioral Happening (see user library in the Happening and Change Model). This gives them the standard behavioral change parts, such as for start and end, so the simple interactions can be ordered within an Interaction Protocol (see the Interaction Protocol Model). This is different from the type of thing transferred.

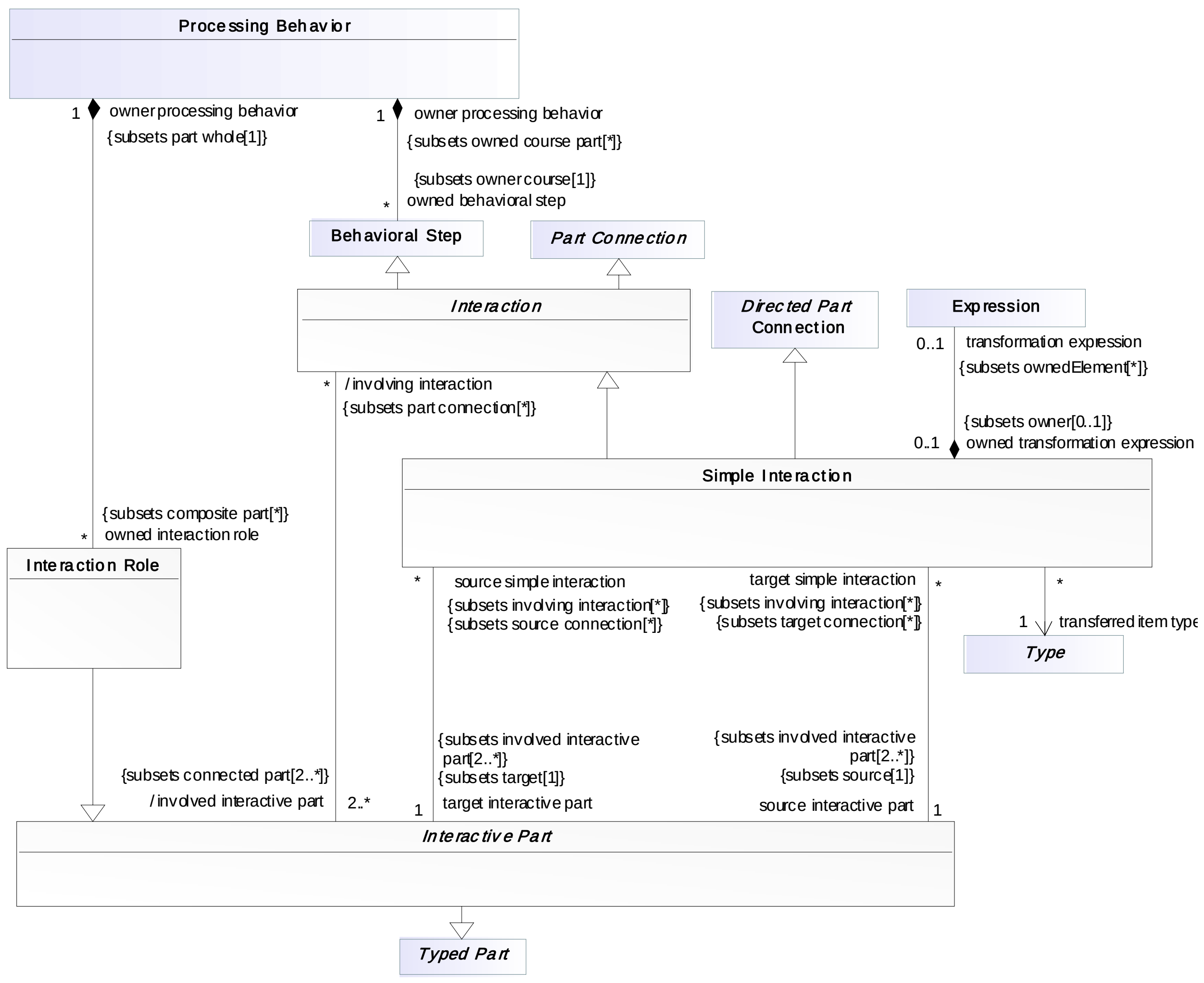
Simple Interaction Bindings are Elements specifying that a simple interaction is the same as some of the simple interactions in the interactive parts it connects. For example, an interaction between steps in a process can be bound to interactions in the connected steps that output and input transferred items (see the Activity Model). The individuals constrained by the binding are interactions as they occur at M0, for example, transferring a car with a certain identification number at a certain time.  These individual (M0) interactions are found by navigating from an individual composite, to individual interactions playing a part in it, and from there to internal interactions in the source end, and to internal interactions in the target end. The three sets of individuals found this way must be exactly the same. Simple interaction bindings are different from connections because interaction bindings are about which individuals are playing certain parts in a whole, whereas connections are about links between the individuals themselves due to playing parts in the whole.

Interaction Roles are Interactive Parts played by individuals outside the processing behavior, but interacting with it.  For example, the customer is an interaction role in a behavior for delivering a product.  This is specialized in other BPDM packages for application to orchestration and choreography (see the Activity Model and the Interaction Protocol Model).

### 6.6.2 Metamodel Specification

The Simple Interaction Model enables interactions to be treated like any other step in a processing behavior, ordered in time, with start and end events.  The model is the basis for flows between process steps and between participants in a choreography (see the Activity Model and the Interaction Protocol Model).  The Simple Interaction Model is the most specialized model in BPDM that still has elements in common between processes and choreographies.

## Simple Interaction



Processing Behavior
1
owner processing behavior
{subsets part whole[1]}
1
owner processing behavior
{subsets owned course part[*]}
{subsets owner course[1]}
*
owned behavioral step
Behavioral Step
Part Connection
Interaction
Directed Part Connection
Expression
0..1
transformation expression
{subsets ownedElement[*]}
*
/ involving interaction
{subsets part connection[*]}
{subsets owner[0..1]}
0..1
owned transformation expression
Simple Interaction
{subsets composite part[*]}
*
owned interaction role
Interaction Role
*
source simple interaction
{subsets involving interaction[*]}
{subsets source connection[*]}
target simple interaction
{subsets involving interaction[*]}
{subsets target connection[*]}
*
*
1
transferred item type
Type
{subsets involved interactive part[2..*]}
{subsets target[1]}
{subsets involved interactive part[2..*]}
{subsets source[1]}
{subsets connected part[2..*]}
/ involved interactive part
2..*
1
target interactive part
source interactive part
1
Interactive Part
Typed Part

Figure 6.53 - Simple Interaction

### 6.6.2.1 Simple Interaction Binding

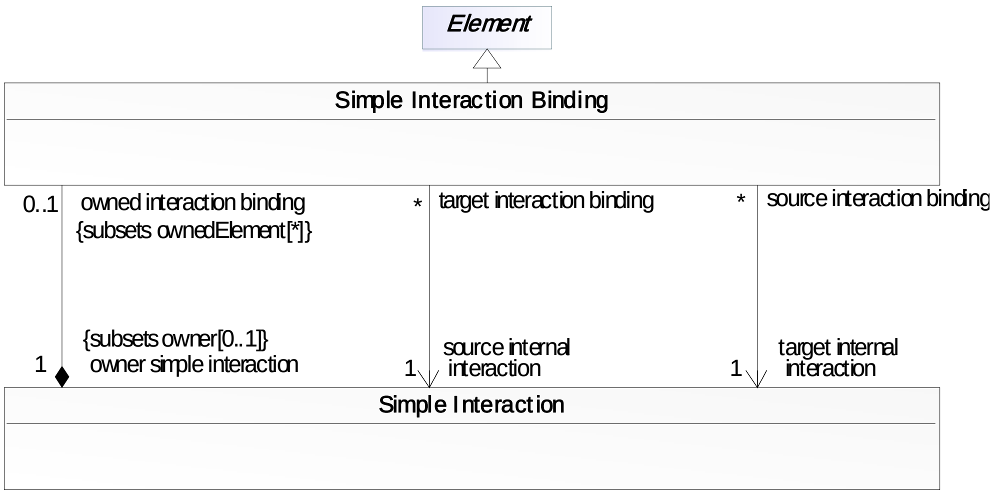

Element
Simple Interaction Binding
0..1
owned interaction binding
{subsets ownedElement[*]}
*
target interaction binding
*
source interaction binding
{subsets owner[0..1]}
1
owner simple interaction
1
source internal interaction
1
target internal interaction
Simple Interaction

**Figure 6.54 - Simple Interaction Binding**

### 6.6.2.2 Interaction

**Namespace:** Simple Interaction
**isAbstract:** Yes
**Generalization:** “Behavioral Step” “Part Connection”

#### Description

An **Interaction** is a **Behavioral Step** that is also a **Part Connection** , enabling **Interaction** to have start and end changes, and be ordered in time.

An **Interaction** can be either a simple **Simple Interaction** or a set of combined **Simple Interaction**s : a **Compound Interaction**. Ultimately, an **Interaction** is realized by the exchange of **Simple Interaction**s between its **Interactive Part**s.

#### Associations

| | |
|---|---|
| involved interactive part : Interactive Part [2..*] | Interactive Part involved in the Interaction<br>Subsets *connected part* |

### 6.6.2.3 Interaction Role

**Namespace:** Simple Interaction
**isAbstract:** No
**Generalization:** “Interactive Part”

#### Description

An **Interaction Role** is an **Interactive Part** where the individuals playing the part are in the environment context where the **Processing Behavior** is used. For example, the customer is an **Interaction Role** in a behavior for delivering a product.

### 6.6.2.4 Interactive Part

**Namespace:** Simple Interaction
**isAbstract:** Yes
**Generalization:** "Typed Part"

#### Description

**Interactive Part** is a category of **Typed Part** that can be connected by **Interactions**. The types of interactive parts establish requirements for the interacting individuals, for example, that they have a minimum security clearance or market capitalization.

#### Associations

| | |
|---|---|
| involving interaction : Interaction [*] | Interaction that the Interactive Part is involved in.<br>Subsets *part connection* |

| | |
|---|---|
| source simple interaction : Simple Interaction [*] | Simple Interaction going to the target interactive part<br>Subsets *involving interaction*<br>Subsets *source connection* |

| | |
|---|---|
| target simple interaction : Simple Interaction [*] | Simple Interaction coming from the source interactive part<br>Subsets *involving interaction*<br>Subsets *target connection* |

### 6.6.2.5 Simple Interaction

**Namespace:** Simple Interaction
**isAbstract:** No
**Generalization:** "Directed Part Connection" "Interaction"

#### Description

A **Simple Interaction** is a kind of **Interaction** in which something is "transferred" from individuals playing one interactive part to individuals playing another interactive part. For example, a document, phone number, or package may be transferred from one department to another in a company. The transferred items must conform to a **Type** specified by the **Simple Interaction**. A **Simple Interaction** can have an **Expression** to change the item that arrives at the target based on the item flowing from the source. For example, a transformation may retrieve the zip code from an address flowing from the source to deliver the zip code to the target.

**Simple Interactions** in user (M1) models are always typed by the **Universal Behavioral Happening** (see user library **Happening & Change Library**). This gives them the standard **Behavioral Change Parts**, such as for start and end, so the **Simple Interactions** can be ordered within an **Interaction Protocol**. This is different from the type of thing transferred.

**Simple Interactions** can refer to **Simple Interactions** inside the **Interactive Parts** being connected. This means the transferred thing is passed along through chains of **Simple Interactions** from inside to outside the parts, or the other way (see **Simple Interaction Binding**)

## Associations

| | |
|---|---|
| owned interaction binding : Simple Interaction Binding [0..1] | Subsets *ownedElement* |

| | |
|---|---|
| source interactive part : Interactive Part [1] | Interactive Part that is the source of the Simple Interaction<br>Subsets *involved interactive part*<br>Subsets *source* |

| | |
|---|---|
| target interactive part : Interactive Part [1] | Interactive Part that is the target of the Simple Interaction<br>Subsets *involved interactive part*<br>Subsets *target* |

| | |
|---|---|
| transferred item type : Type [1] | specifies the type of the item transferred by the Simple Interaction. |

| | |
|---|---|
| transformation expression : Expression [0..1] | Expression used to transform the item that arrives at the target based on the item flowing from the source. For example, a transformation may retrieve the zip code from an address flowing from the source to deliver the zip code to the target.<br>Subsets *ownedElement* |

## BPMN Notation

An **Artifact Sequence Flow** is represented by a is line with a solid arrowhead that MUST be drawn with a solid single line.

The type of the element transferred by the information flow is represented by a portrait-oriented rectangle that has its upper-right corner folded over that MUST be drawn with a solid single black line.

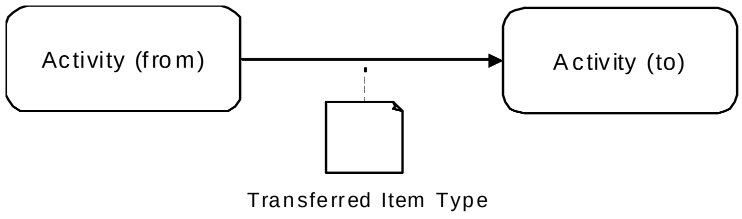


**Figure 6.55 - Artifact Sequence Flow Notation**

Notation for **End Message** or **Simple Interaction** categorized as an **End Message**.

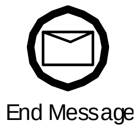


**Figure 6.56 - End Message Notation**

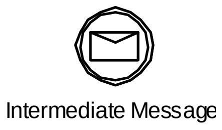


**Figure 6.57 - Intermediate Message Notation or Change Condition Step Monitoring an incoming Message**

A **Message Flow** is line with a open arrowhead that MUST be drawn with a dashed single black line.

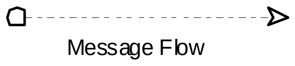


**Figure 6.58 - Message Flow Notation**

The shape of **Message** depends on its sub-types.

The line connecting a **Message** to its **Interaction Role**(s) MUST have an open arrowhead and MUST be drawn with a dashed single black line.

The line connecting a **Message** to other kind of **Interactive Part** MUST have a solid arrowhead and MUST be drawn with a solid single line.

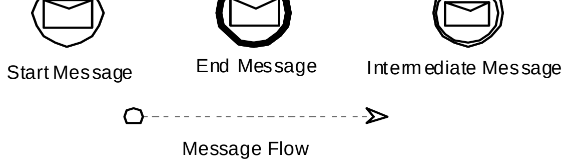


**Figure 6.59 - Message Notation**

Notation for **Start Message** or **Simple Interaction** categorized as a **Start Message**.

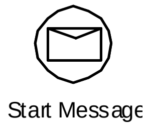


**Figure 6.60 - Start Message Notation**

### Non-normative Notation

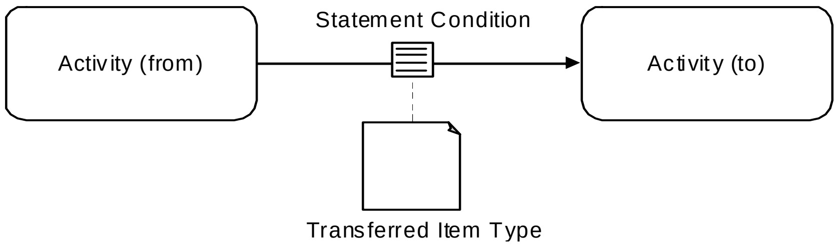


**Figure 6.61 - Interaction Flow between Activities and Statement Condition**

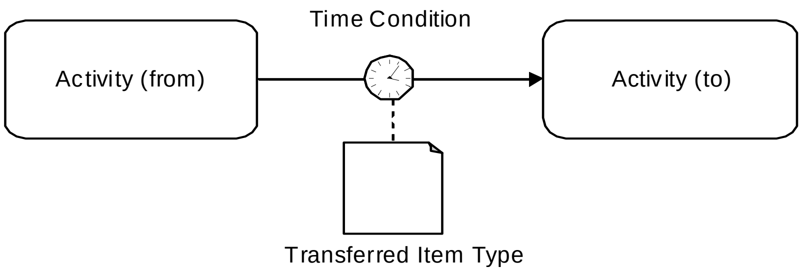


**Figure 6.62 - Interaction Flow between Activities and Time Condition**

## 6.6.2.6 Simple Interaction Binding

**Namespace:** Simple Interaction
**isAbstract:**
**Generalization:** “Element”

### Description

**Simple Interaction Binding** is a kind of **Element** specifying that a **Simple Interaction** is the same as some of the **Simple Interactions** in the **Interactive Parts** it connects. The individuals constrained by the binding are **Simple Interactions** as they occur at M0, for example, transferring a car with a certain identification number at a certain time. These individual (M0)**Simple Interactions** are found by navigating from an individual composite, to individual interactions playing a part in it, and from there to **source internal interaction** in the source ,**Interactive Part** and to **source internal interaction** in the target **Interactive Part**. The three sets of individuals found this way must be exactly the same. **Simple Interaction Bindings** are different from **Part Connections** because interaction bindings are about which individuals are playing certain parts in a whole, whereas connections are about links between the individuals themselves due to playing parts in the whole.

**Associations**

| | |
|---|---|
| source internal interaction : Simple Interaction [1] | Simple Interaction played in the source interactive part |

| | |
|---|---|
| target internal interaction : Simple Interaction [1] | Simple Interaction Binding played in the target interactive part |

**Constraint**

[1] The **target internal interaction** must be one of the **Simple Interactions** of the **Behavioral Happening** that is the type of the **source interactive part**.

> self.**target internal interaction** in self.**source interactive part** ->**process type** ->**owned interaction role** ->**target simple interaction**

## 6.7 Activity Model

### 6.7.1 Introduction

The Activity Model is for capturing orchestrations in way that facilitates modification as boundaries of process of business change, for example, due to insourcing, outsourcing, mergers, and acquisitions. It uses interactions to represent inputs and outputs, enabling choreographies to be specified between the process and its environment, as well as between the performers responsible for steps in the process. The Activity Model is the basis for the BPMN model in BPDM (see the BPMN Extension).

In the Activity Model, Processes are Processing Behaviors that have:

- Boundaries with which processes interact to get inputs and provide outputs (Process Interaction Boundary).
- Performers for steps in the process, including a performer for the entire process (Performer Role and Processor Role).
- Steps that interact with each other and the process boundary, and invoke other processes (Activity and Emdedded Process).
- Embedded processes for loops, with loop control features (Activity Loop and its subtypes).
- Holders hold flowing items (Holders).
- Steps for generating process lifecycle events, such as for errors and aborts.
- Derivations from other processes (Substitutable Derivations).

Process Interaction Boundaries and Processor Roles are the two top-level elements in Processes. The first represents entities in the environment of the process and the other the actors responsible for the process itself. They are Interactive Parts, enabling Simple interactions between them to show the inputs and outputs of a process (see the Simple Interaction Model). Inputs are simple interactions that have the boundary as source and the processor as target (or an activity in the processor, see below), and outputs have the processor as source (or an activity in the processor), and the boundary as target. The transferred item type of simple interactions specifies the kind of thing that is input or output. These interactions can be ordered in time to specify when the process is expecting its inputs and when it will provide its outputs. Multiplicities on the interactions specify how many individuals of the item type are required or allowed to be input and output by the process (see the Composition Model).

Performer Roles are Part Groups showing the responsibility of Actors for steps in the process (see Activity below). Processor Roles are actually just top-level Performer Roles, enabling them to delegate responsibility for a subset of the process steps to Performer Roles, which in can turn delegate smaller subsets to other Performer Roles. Processor Roles and Performer Roles are also Typed Parts, for specifying Actors that can play the roles. Actors are Types, to specify requirements on them, such as having certain skills or budget.

Performer Roles are also Interactive Parts that can have interactions with each other as well to the boundary. This is useful when the boundaries of the process change, for example, due to outsourcing or insourcing. For outsourcing, the steps a performer role is responsible for are separated out into another process. The interactions between the performer's steps and the steps of other performers become the interactions in the protocol between the performers. This establishes a service contract for the outsourced steps in the activity. Role Realizations are Elements for showing which processes satisfy the contract. For insourcing, some of the interactions to the boundary become interactions with a performer role. This establishes the requirements on designing the steps that the performer will be responsible for.

Activities are:

- Processing Steps, enabling them to have start and end changes, be ordered in time by processing successions, and nest subprocesses (see the Processing Behavior Model).
- Typed Parts (due to being Processing Steps), where the type is another Process. For Simple Activities the subprocesses have no subactivities, for Subprocess Activities they do.
- Interactive Parts to support simple interactions with other activities and the boundary for inputs and outputs (see the Simple Interaction Model).

Activities connected by Simple Interactions use Simple Interaction Bindings to specify which interactions in the subprocesses will flow between the activities (see the Simple Interaction Model). For example, one activity might be for a process that outputs a document with an interaction to its boundary, and another activity might be for a process that inputs a document with an interaction from its boundary. These process might output and input many other documents. The simple interaction bindings on the interaction between the activities identify which of the interactions in the subprocesses are the ones that support the flow between the activities. The bindings ensure that whenever the document flows during the enactment of the process, that the exact same M0 flow plays all three interaction parts simultaneously: the output interaction in one subprocess, the interaction between the activities, and the input interaction in the other subprocess. In many cases, the simple interaction bindings can be derived from the types of things flowing, so the modeler does not need to specify them manually. For example, if the subprocesses have only one interaction outputting and inputting a document, then interaction flows transferring documents between the subprocess will bind to those internal interaction.[11]

Emdedded Processes are Behavioral Step Groups that enclose Activities, enabling embedded processes to have their own lifecycle changes, such as starting and ending, that interact with the enclosed activities. Every embedded process has the Abort Group Connection applied to it (see Processing Behavior). This ensures the enclosed steps abort when the group does.

Activity Loops are Emdedded Processes that can execute their enclosed activities as a group multiple times. The process can proceed past the loop in several ways:

- After all subexecutions are complete, with a processing succession that has the loop as the source.

11. Simple interaction bindings can be derived if the interaction between the activities has a transferred item type that is the same or a supertype of exactly one output interaction flow on the source end of the interaction, or has a transferred item type that is the same or a subtype of exactly one input interaction flow on the target end of the interaction.

- After each subexecution, with processing succession that has the iterationEnd behavior part as an internal source. This part is defined in a user (M1) library in the Activity Model, typed by the IterationEnd change also defined in the library.
- After the first subexecution to complete, with a processing succession that has the iterationEnd behavior part as an internal source, and a guard evaluating to the string “first iteration.”
- After each subexecution, but depending on conditions, with a processing succession that has the iterationEnd behavior part as an internal source, and a guard specified by the modeler.

Activity Loops are of two kinds:

- Conditional Loops execute their enclosed activities multiple times as a group while a specified condition is true. If the condition is never true, the enclosed activities are never executed. The multiple subexecutions are sequential.
- MultiInstance Loops execute their enclosed activities as a group a certain number of times, as specified by the modeler in an integer-valued expression evaluated at the time the loop begins executing. MultiInstance Loops support the option of sequential or parallel subexecutions.

Holders are Interactive Parts for storing items temporarily as they flow through the process. For example, a document, phone number, or package can flow along simple interactions, into a holder for some period, and flow out later. The type of the holder is the type of thing it can hold.

Substitutable Derivations are Derivations of one process from another that do not alter the interactions with the boundary (see the Composition Model).

### 6.7.2 Metamodel Specification

The Activity Model is for capturing orchestrations in way that facilitates modification as boundaries of process of business change, for example, due to insourcing, outsourcing, mergers, and acquisitions. It uses interactions to represent inputs and outputs, enabling choreographies to be specified between the process and its environment, as well as between the performers responsible for steps in the process. The Activity Model is the basis for the BPMN model in BPDM (see the BPMN Extensions).

### 6.7.2.1  Core

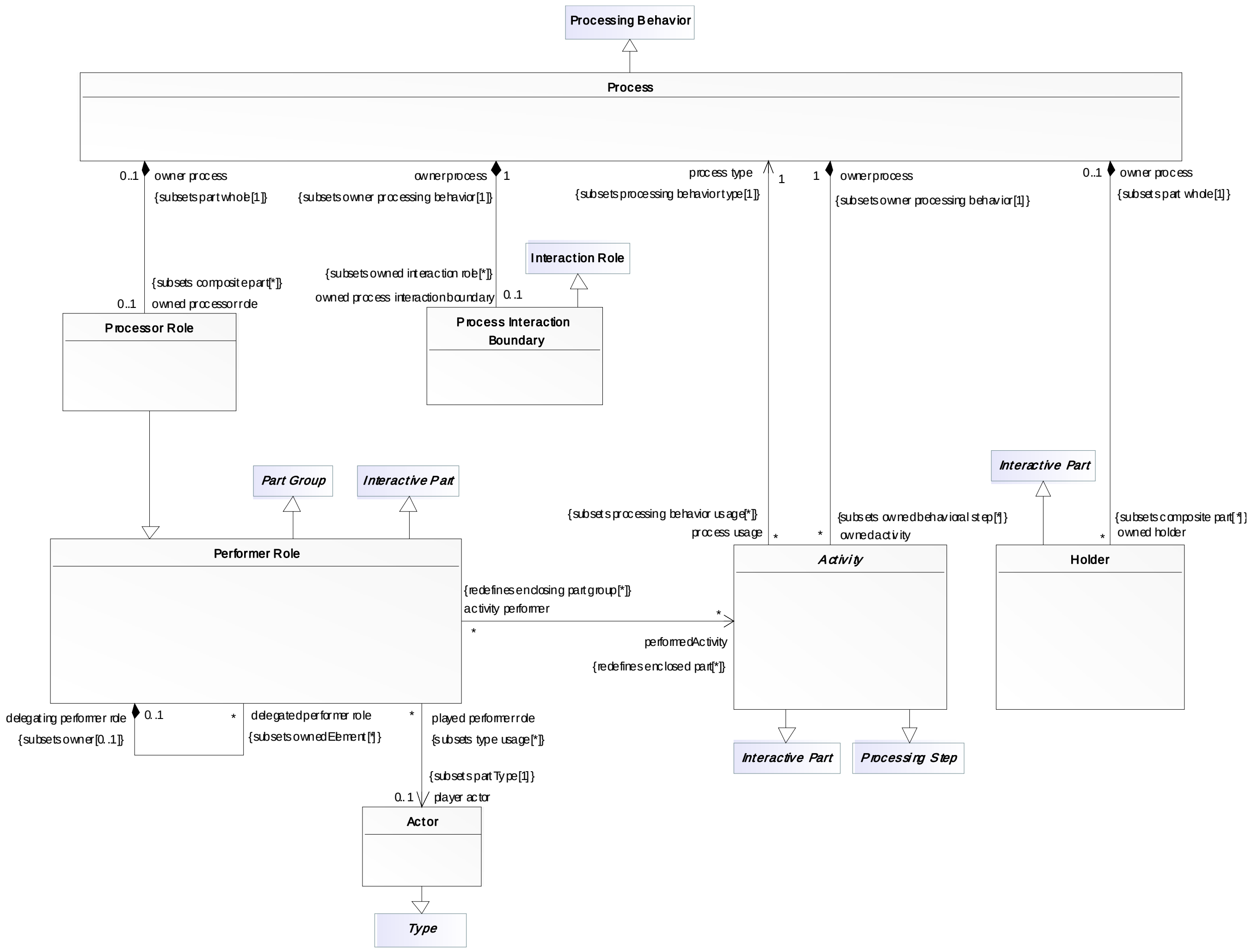


**Figure 6.63 - Core**

### 6.7.2.2 Activity Model Library: Simple Process instances

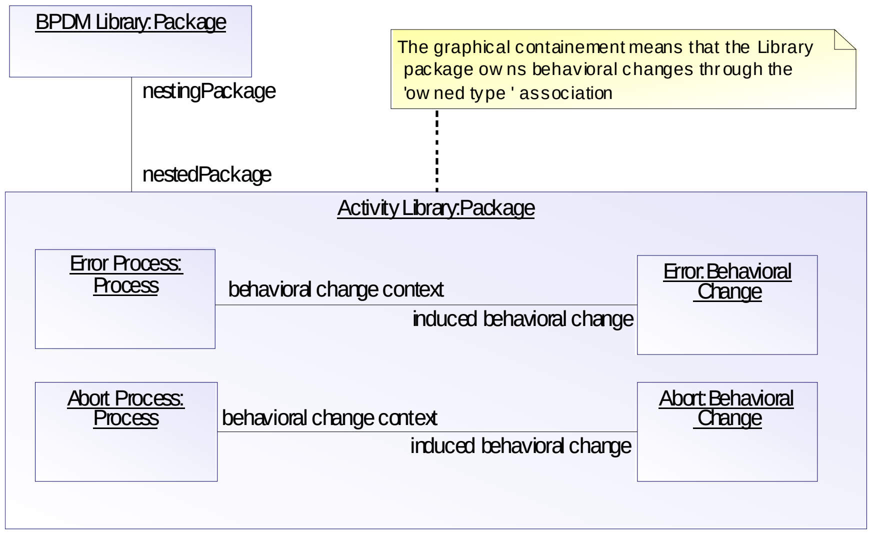

BPDM Library:Package
nestingPackage
nestedPackage
The graphical containement means that the Library package owns behavioral changes through the 'owned type' association
Activity Library:Package
Error Process: Process
behavioral change context
induced behavioral change
Error:Behavioral Change
Abort Process: Process
behavioral change context
induced behavioral change
Abort: Behavioral Change

**Figure 6.64 - Activity Model Library: Simple Process instances**

### 6.7.2.3 Activity Categories

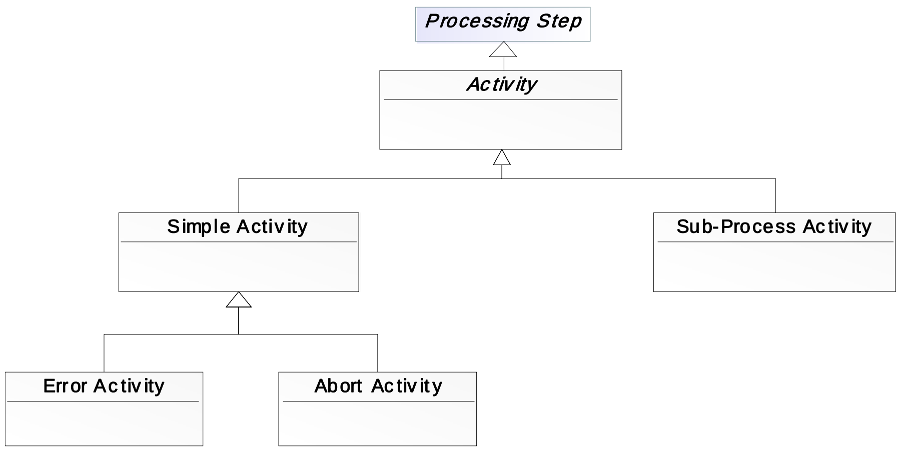


**Figure 6.65 - Activity Categories**

### 6.7.2.4 Activity Model Library: Loop Happening instance

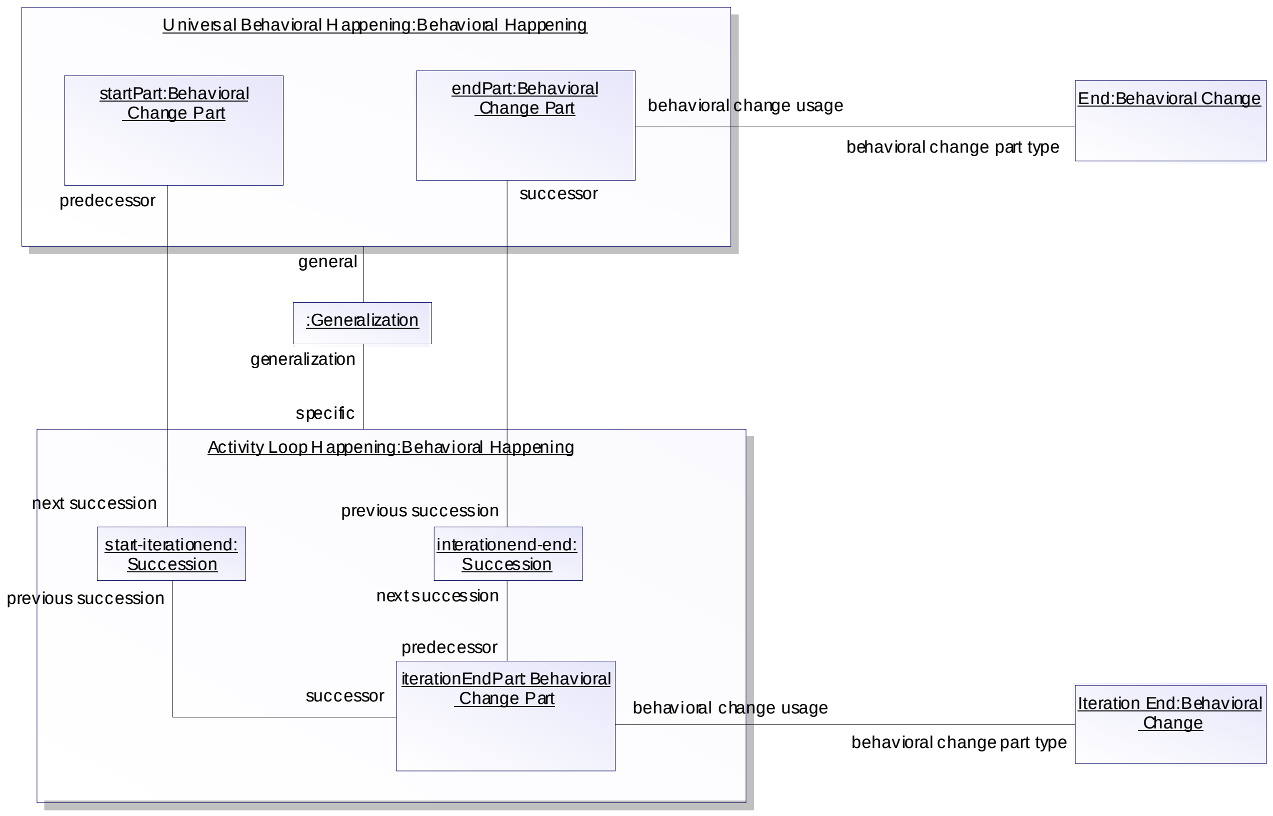

Universal Behavioral Happening:Behavioral Happening
startPart:Behavioral Change Part
endPart:Behavioral Change Part
behavioral change usage
End:Behavioral Change
behavioral change part type
predecessor
successor
general
:Generalization
generalization
specific
Activity Loop Happening:Behavioral Happening
next succession
previous succession
start-iterationend: Succession
interationend-end: Succession
previous succession
next succession
predecessor
successor
iterationEndPart Behavioral Change Part
behavioral change usage
Iteration End:Behavioral Change
behavioral change part type

Figure 6.66 - Activity Model Library: Loop Happening instance

## 6.7.2.5 Embedded Process

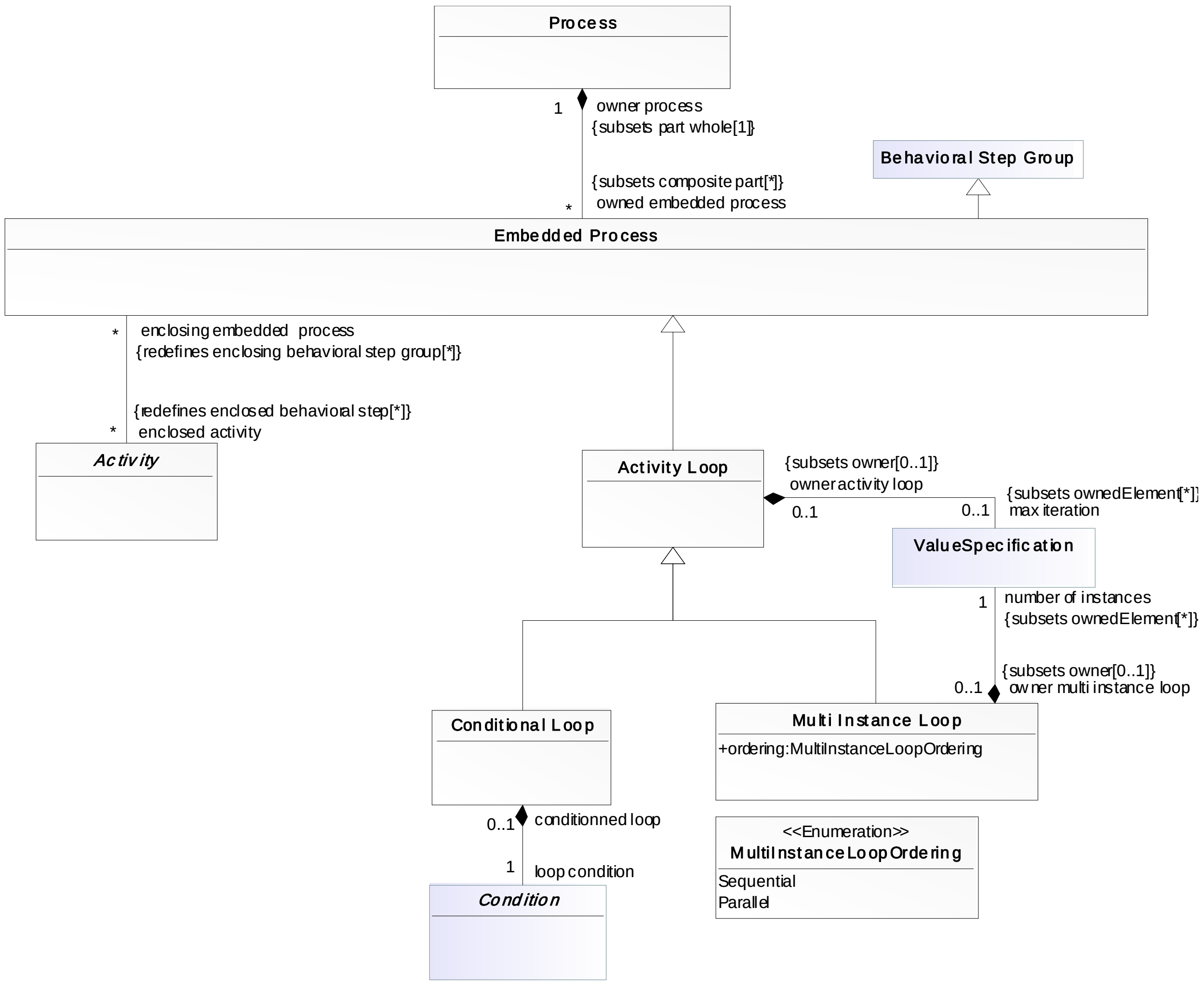


**Figure 6.67 - Embedded Process**

## 6.7.2.6 Derivation

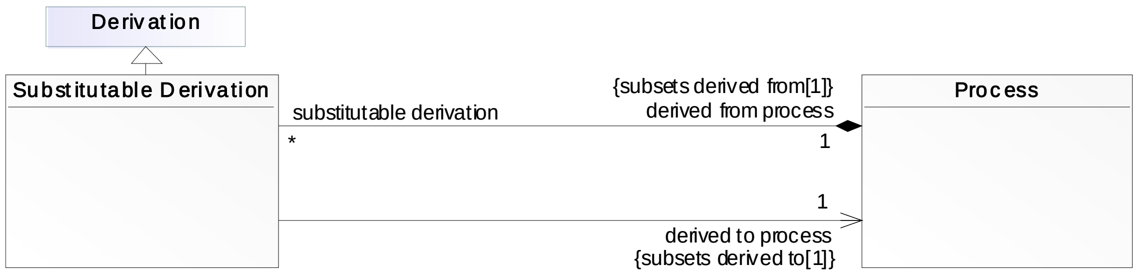


**Figure 6.68 - Derivation**

### 6.7.2.7 Role Realization

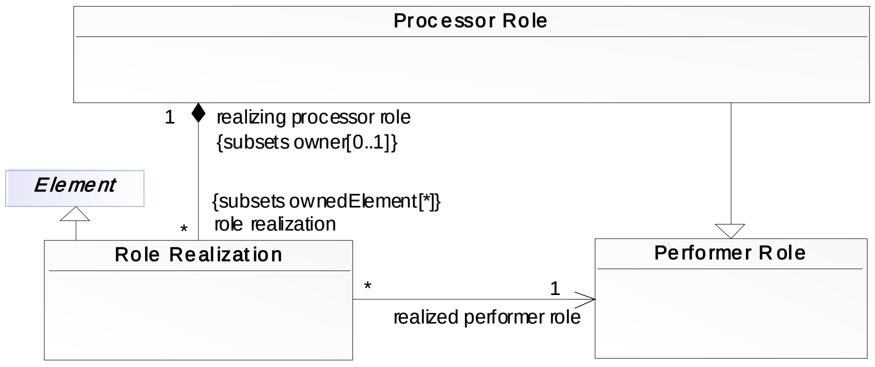


**Figure 6.69 - Role Realization**

### 6.7.2.8 Abort Activity

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Simple Activity”

#### Description

An **Abort Activity** is a **Simple Activity** that interrupts the course of a **Process**. All activities in the **Process** should be immediately ended. The **Process** is ended without compensation or event handling.

The type of all **Abort Activity(ies)** must be **Abort Process** provided by BPDM user library for the Activity Model (**Activity Library**).

#### BPMN Notation

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Abort** instance of **Behavioral Change**
2. An **Abort Activity**

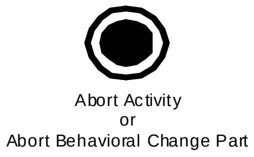


**Figure 6.70 - Abort Activity Notation or 'Abort' Behavioral Change Part**

### 6.7.2.9 Activity

**Namespace:** Activity Model
**isAbstract:** Yes
**Generalization:** “Interactive Part” “Processing Step”

### Description

An **Activity** is a kind of **Processing Step** that activates a **Processing Behavior** (it operates over time) in the context of a **Process**. It can:

- be ordered in time by **Processing Succession**
- operate under the responsibility of a **Performer Role**
- activate a sub-processe or be a simple task that start and stop

An **Activity** is also an **Interactive Part** that receives its inputs and outputs through **Interactions** coming from other **Interactive Part**s in the **Process** (**Activity**, **Interaction Role**, **Performer Role**, **Holder**).

### Associations

| | |
|---|---|
| enclosing embedded process : Embedded Process [*] | Embedded Process in which the activity is enclosed<br>Redefines *enclosing behavioral step group* |

| | |
|---|---|
| process type : Process [1] | Type of the Activity<br>Subsets *processing behavior type* |

### BPMN Notation

An Activity is represented by a rounded corner rectangle that MUST be drawn with a single thin black line.

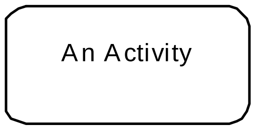


**Figure 6.71 - Activity Notation**

## 6.7.2.10 Activity Loop

**Namespace:** Activity Model
**isAbstract:**
**Generalization:** “Embedded Process”

### Description

An **Activity Loop** is an **Embedded Process** that can execute its their enclosed activities multiple times. The process can proceed past the loop in several ways:

- After all subexecutions are complete, with a processing succession that has the loop as the source.
- After each subexecution, with processing succession that has the iterationEnd behavior part as an internal source. This part is defined in a user (M1) library in the Activity Model, typed by the IterationEnd change also defined in the library.
- After the first subexecution to complete, with a processing succession that has the **iterationEndPart** as an internal source, and a guard evaluating to the string “first iteration.”

- After each subexecution, but depending on conditions, with a processing succession that has the iterationEnd behavior part as an internal source, and a guard specified by the modeler.

### Associations

| | |
|---|---|
| max iteration : ValueSpecification [0..1] | the maximum number of iteration<br>Subsets *ownedElement* |

## 6.7.2.11 Actor

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Type”

### Description

An **Actor** is an entity that is responsible for the execution of duties specified by a **Performer Role**

Further sub-type of **Actor** will be defines in specifications such as the the Organizational Structure Metamodel (OSM) to add specific requirements such as  and can as having certain skills or budget.

## 6.7.2.12 Conditional Loop

**Namespace:** Activity Model
**isAbstract:**
**Generalization:** “Activity Loop”

### Description

**Conditional Loop** is a kind of **Activity Loop** that will execute its enclosed activities multiple times as a group while a specified condition is true. If the condition is never true, the enclosed activities are never executed. The multiple subexecutions are sequential.

### Associations

| | |
|---|---|
| loop condition : Condition [1] | Condition that controls the iterations of a Conditional Loop |

## 6.7.2.13 Embedded Process

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Behavioral Step Group”

### Description

An **Embedded Process** is a kind of **Behavioral Step Group** that groups a set of **Activity** that, as a whole, act as a **Processing Step**. Thereby, an **Embedded Process** is typed by a **Behavioral Happening** that defines its **start change** and a **finish change**.

As any **Processing Step,** an **Embedded Process** can be interrupted or constrained in its **Behavioral Happening** course.

### Associations

| | |
|---|---|
| enclosed activity : Activity [*] | Activity that is part of the Embedded Process<br>Redefines *enclosed behavioral step* |

### Constraint

[1] An **enclosed activity** of an **Embedded Process** must belong to the **Process** owning the **Embedded Process**

### BPMN Notation

A Sub-Process Activity shares the same shape as the Activity object, which is a rounded rectangle. A Sub-Process Activity is a rounded corner rectangle that MUST be drawn with a single thin black line. If the Sub-Process Activity is also a transaction, it has a boundary drawn with a double line.

The Sub-Process Activity can be in a collapsed view that hides its details or a Sub-Process can be in an expanded view that shows the details of its Process Type.

In the collapsed form, the Sub-Process Activity uses a marker to distinguish it as a Sub-Process Activity, rather than a Simple Activity. The Sub-Process Activity marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.

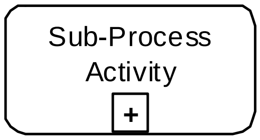


**Figure 6.72 - Collapsed Sub-Process Activity Notation**

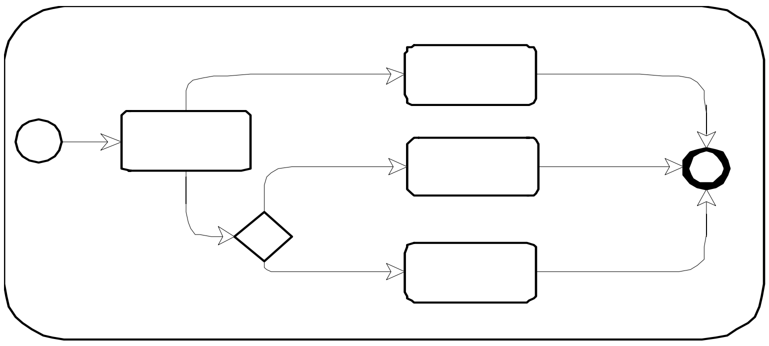


**Figure 6.73 - Uncollapsed Sub-Process Activity Notation**

### 6.7.2.14 Error Activity

**Namespace:** Activity Model
**isAbstract:**

**Generalization:** “Simple Activity”

### Description

An **Error Activity** is a kind of **Simple Activity** that produces an **Error** and that ends its enclosing **Behavioral Happening**

In case where the **Error Activity** is part of an **Embedded Process**, the ended **Behavioral Happening** is this **Embedded Embedded Process**, otherwise the ended **Behavioral Happening** is the **Process** that owns the **Error Activity**.

### BPMN Notation

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Error** instance of **Behavioral Change**
2. An **Error Activity**

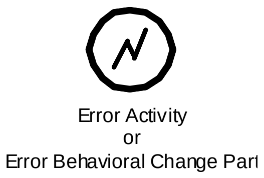


**Figure 6.74 - Error Activity Notation or 'Error' Behavioral Change Part**

## 6.7.2.15 Holder

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Interactive Part”

### Description

A **Holder** is an **Interactive Part** storing items temporarily as they flow through the **Process**. For example, a document, phone number, or package can flow along simple interactions, into a holder for some period, and flow out later. The type of the **Holder** is the type of thing it can hold.

### Non-normative Notation

A Holder is represented by a can that MUST be drawn with a single thin black line.

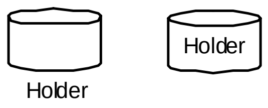


**Figure 6.75 - Holder Notation**

## 6.7.2.16 LoopTestTime

**Namespace:** Activity Model
**isAbstract:**

### Description

Enumeration of the following literal values:

| | |
|---|---|
| after: | |

| | |
|---|---|
| before: | |

### 6.7.2.17 Multi Instance Loop

**Namespace:** Activity Model
**isAbstract:**
**Generalization:** “Activity Loop”

### Description

**Multi Instance Loop** is a kind of **Activity Loop** that will execute its enclosed activities as a group  of times, as specified by the **number of instances ValueSpecification** evaluated at the time the loop begins executing.  A **Multi Instance Loop** supports the option of sequential or parallel subexecutions as specified by its **ordering** attribute.

### Attributes

| | |
|---|---|
| ordering: MultiInstanceLoopOrdering [] | |

### Associations

| | |
|---|---|
| number of instances : ValueSpecification [1] | number of instance of iteration<br>Subsets *ownedElement* |

### 6.7.2.18 MultiInstanceLoopOrdering

**Namespace:** Activity Model
**isAbstract:**

### Description

Enumeration of the following literal values:

| | |
|---|---|
| Parallel: | activities are executed in parallel |

| | |
|---|---|
| Sequential: | the activities are executed sequentially |

### 6.7.2.19 Performer Role

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Interactive Part” “Part Group”

### Description

A **Performer Role** is a **Part Group** that takes responsibility of performing activities in the process. Being an **Interactive Part**, a **Performer Role** also has responsibilities to fulfill **Interactions** that it is involved with other **Performer Roles** or with **Interaction Roles** at the boundary of the **Process**. A **Performer Role** is a **Typed Part** for specifying **Actor** that can play the role at process enactment.

A **Performer Role** can be decomposed into sub **Performer Role** to delegate responsibility for a subset of its activities or interactions. A **Performer Role** may have a realization as defined by a **Role Realization** that further specifies how the **Performer Role** will meet its responsibilities.

### Associations

| | |
|---|---|
| performedActivity : Activity [*] | specifies the set of Activity(ies) that are under the responsibility of the Performer Role<br>Redefines *enclosed part* |

| | |
|---|---|
| player actor : Actor [0..1] | Actor that, at runtime, is responsible for the execution of the responsibilities specified by the Performer Role<br>Subsets *partType* |

### BPMN Notation

A Performer Role is represented by a Lane. A lane is a sub-partition of the Pool representing the **Processor Role** of the process or a sub-partition of the Lane representing its delegating performer role.

A Lane will extend the entire length of its containing Pool or Lane, either vertically or horizontally . If the pool is invisibly bounded, the lane associated with the pool must extend the entire length of the pool.

Text associated with the Lane (the Performer Role name) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Our examples place the name as a banner on the left side (for horizontal Pools) or at the top (for vertical Pools) on the other side of the line that separates the Pool name, however, this is not a requirement.

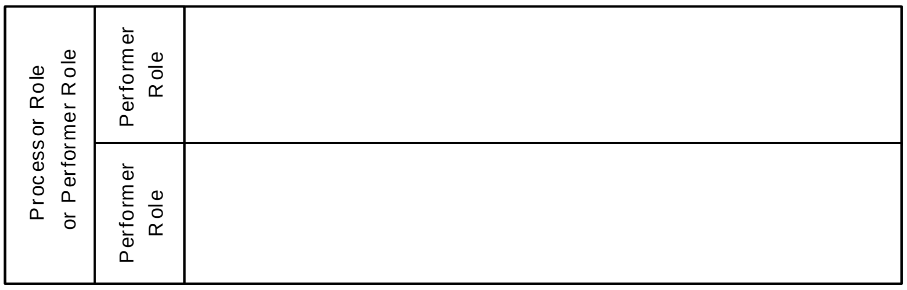


Figure 6.76 - Horizontal Lane Notation

A Performer Role is represented by a Lane. A lane is a sub-partition of the Pool representing the Processor Role of the process or a sub-partition of the Lane representing its delegating performer role.

A Lane will extend the entire length of its containing Pool or Lane, either vertically or horizontally . If the pool is invisibly bounded, the lane associated with the pool must extend the entire length of the pool.

Text associated with the Lane (the Performer Role name) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Our examples place the name as a banner on the left side (for horizontal Pools) or at the top (for vertical Pools) on the other side of the line that separates the Pool name, however, this is not a requirement.

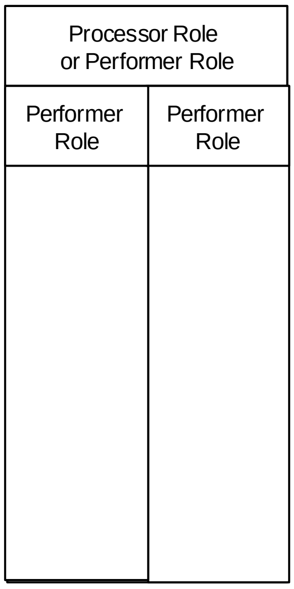


**Figure 6.77 - Vertical Lane Notation**

### 6.7.2.20 Process

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** "Processing Behavior"

#### Description

A **Process** is a kind of **Processing Behavior** that describes specific **Activity**(ies) to be performed, **Interaction**s to be undertaken during its execution under the authority of a **Processor Role** (or **delegated performer role** s).

The process owns the set of activities to be performed as well as the **Condition**s on when such activities will be performed and by which performer role. The process also owns the set of **Interactive Part**s that define the flow of information and other resources between activities,**Performer Role** and **Interaction Role**s.

A specific **Interaction Role** defines the set of **Interaction**s the process is responsible of: its is the **Process Interaction Boundary**. The set of **Interaction**s attached to the **Process Interaction Boundary** defines the inputs and outputs of the process

A Process may utilize sub-processes with a **Sub-Process Activity** as well as be used in the context of other processes in the same way.

## Associations

| | |
|---|---|
| owned activity : Activity [*] | Activity owned by the Process<br>Subsets *owned behavioral step* |

| | |
|---|---|
| owned embedded process : Embedded Process [*] | Embedded Process owned by the Process<br>Subsets *composite part* |

| | |
|---|---|
| owned holder : Holder [*] | Holder owned by the Process<br>Subsets *composite part* |

| | |
|---|---|
| owned process interaction boundary : Process Interaction Boundary [0..1] | specifies the set of Interactions the process is responsible for.<br>This set of Interaction defines the inputs and outputs of the process<br>Subsets *owned interaction role* |

| | |
|---|---|
| owned processor role : Processor Role [0..1] | Processor Role of the Process<br>Subsets *composite part* |

| | |
|---|---|
| substitutable derivation : Substitutable Derivation [*] | |

## Non Normative Notation

Each process diagram has a contents area. As an option, it may have a frame and a heading as shown in the following figure. The frame is a rectangle. The frame may be omitted and implied by the border of the diagram area provided by a tool. In case the frame is omitted, the heading is also omitted.

The diagram contents area contains the graphical symbols. The heading is a string contained in name tag (rectangle with cutoff corner) in the upper leftmost corner of the rectangle, with the following syntax: <process name>

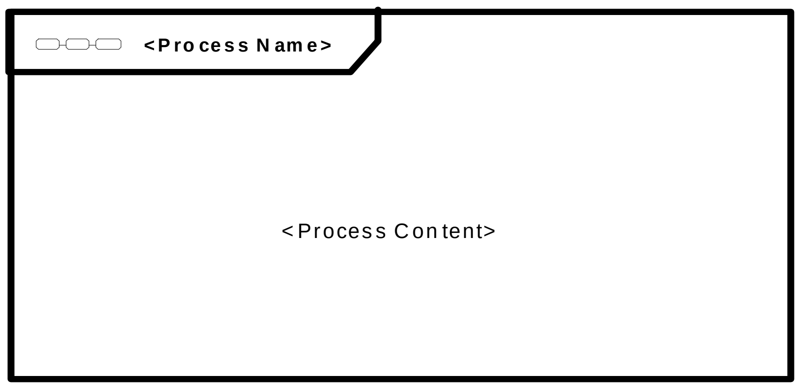


**Figure 6.78 - Process Diagram**

### 6.7.2.21 Process Interaction Boundary

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Interaction Role”

#### Description

The **Process Interaction Boundary** is the **Interaction Role** through which a **Process** interacts to get its inputs and deliver its outputs.

The process is responsible to fulfill all **Interactions** attached to the **Process Interaction Boundary**.

### 6.7.2.22 Processor Role

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Performer Role”

#### Description

A **Processor Role** is the top-level **Performer Role** reponsible for all activities and interactions at the boundary of the **Process**. As all **Performer Roles**, it can delegate responsibility for a subset of the process activities and interactions to **Performer Roles**, which in can turn delegate smaller subsets to other **Performer Roles** (**delegated performer role** ).

A **Processor Role** may be active or passive. An active processor will control and/or monitor the process and may manage process resources. A passive processor delegates all responsibilities to delegee role. The actor of a passive processor may be a “community”, consensus body or group of actors who have agreed to work together in a particular way. The actor of an active processor must be an individual, system or organization capable of taking action, initiating and responding to **Interactions** and managing resources.

#### Associations

| | |
|---|---|
| role realization : Role Realization [*] | specification of the set of Performer Role that the Processor Role is the realization of<br>Subsets *ownedElement* |

#### BPMN Notation

A Processor Role is represented by a Pool. A Pool is a square-cornered rectangle that MUST be drawn with a solid single black line.

To help with the clarity of the Diagram, A Pool will extend the entire length of the Diagram, either horizontally or vertically. However, there is no specific restriction to the size and/or positioning of a Pool. Modelers and modeling tools can use Pools (and Lanes) in a flexible manner in the interest of conserving the “real estate” of a Diagram on a screen or a printed page.

The Processor Role Pool MAY be presented without a boundary.

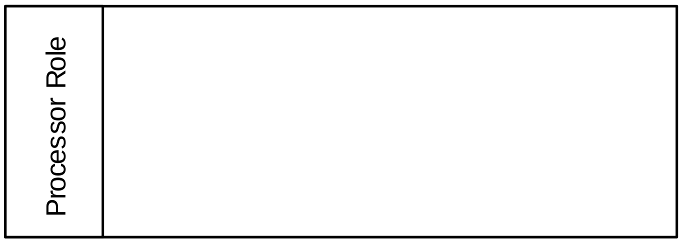


**Figure 6.79 - Processor Role Notation**

### 6.7.2.23 Role Realization

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Element”

#### Description

A role realization takes a realized performer role and defines a processor role and the associated process that specifies the specific process to be enacted by the specified processor role as required to meet the responsibilities of the realized performer role. A performer role may be realized by any number of processor roles as long as they each satisfy the responsibilities of the role.

#### Associations

| | |
|---|---|
| realized performer role : Performer Role [1] | Performer Role that is the specification of the Role Realization. |

### 6.7.2.24 Simple Activity

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Activity”

#### Description

A **Simple Activity** is an **Activity** which **process type** is no further composed of other activities

#### Constraint

[1] A **Simple Activity** is typed by a process that has no owned activity

self.oclIsKindof(**Simple Activity**) implies self.**process type** ->**owned activity** ->isEmpty()

#### BPMN Notation

An Activity is represented by a rounded corner rectangle that MUST be drawn with a single thin black line.

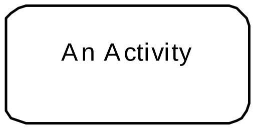


**Figure 6.80 - Activity Notation**

### 6.7.2.25 Sub-Process Activity

**Namespace:** Activity Model
**isAbstract:** No
**Generalization:** “Activity”

#### Description

A **Sub-Process Activity** is an **Activity** which **process type** is further composed of other activities

#### Constraint

[1] A **Sub-Process Activity** is typed by a process that has owned activity

self.oclIsKindOf(**Activity**) implies self.**process type** ->**owned activity** ->notEmpty()

#### BPMN Notation

A Sub-Process Activity shares the same shape as the Activity object, which is a rounded rectangle. A Sub-Process Activity is a rounded corner rectangle that MUST be drawn with a single thin black line. If the Sub-Process Activity is also a transaction, it has a boundary drawn with a double line.

The Sub-Process Activity can be in a collapsed view that hides its details or a Sub-Process can be in an expanded view that shows the details of its Process Type.

In the collapsed form, the Sub-Process Activity uses a marker to distinguish it as a Sub-Process Activity, rather than a Simple Activity. The Sub-Process Activity marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.

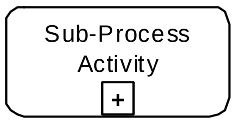


**Figure 6.81 - Collapsed Sub-Process Activity Notation**

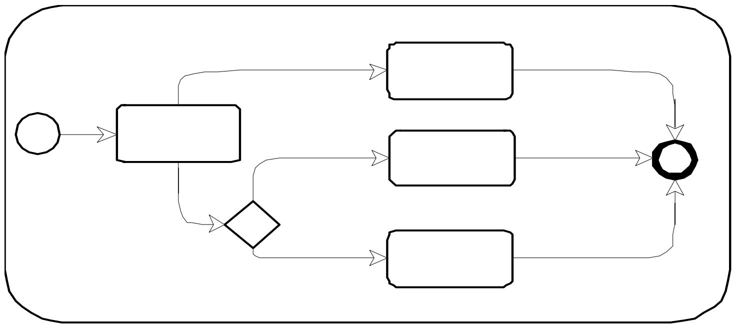


**Figure 6.82 - Uncollapsed Sub-Process Activity Notation**

#### 6.7.2.26 Substitutable Derivation

**Namespace:** Activity Model

**isAbstract:**

**Generalization:** “Derivation”

##### Description

A **Substitutable Derivation** is a kind of **Derivation** that derives one **Process** from another and that does not alter the **Interaction** at the **owned process interaction boundary**

##### Associations

| | |
|---|---|
| derived to process : Process [1] | Subsets *derived to* |

#### 6.7.2.27 Instance: Abort Process

**Class:** Process

##### Description

##### Links

| | |
|---|---|
| Played End | *Opposite End* |
| behavioral change context | *induced behavioral change* Abort |
| ownedType | *package* Activity Library |

#### 6.7.2.28 Instance: Activity Library

**Class:** Package

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| nestedPackage | *nestingPackage* BPDM Library |
| nestingPackage | *nestedPackage* Compensation Library |
| package | *ownedType* Error Process |
| package | *ownedType* Abort Process |
| package | *ownedType* Activity Loop Happening |
| package | *ownedType* Iteration End |

### 6.7.2.29 Instance: Activity Loop Happening

**Class:** Behavioral Happening

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| ownedType | *package* Activity Library |
| owner behavioral happening | *owned behavioral change part* iterationEndPart |
| owner course | *owned succession* start-iterationend |
| owner course | *owned succession* interationend-end |
| specific | *generalization* Generalization |

### 6.7.2.30 Instance: Error Process

**Class:** Process

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change context | *induced behavioral change* Error |
| ownedType | *package* Activity Library |

### 6.7.2.31 Instance: Generalization

**Class:** Generalization

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| | general Universal Behavioral Happening |
| generalization | *specific* Activity Loop Happening |

### 6.7.2.32 Instance: interationend-end

**Class:** Succession

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* iterationEndPart |
| owned succession | *owner course* Activity Loop Happening |
| previous succession | *successor* endPart |

### 6.7.2.33 Instance: Iteration End

**Class:** Behavioral Change

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* iterationEndPart |
| ownedType | *package* Activity Library |

### 6.7.2.34 Instance: iterationEndPart

**Class:** Behavioral Change Part

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Iteration End |
| owned behavioral change part | *owner behavioral happening* Activity Loop Happening |
| predecessor | *next succession* interationend-end |
| successor | *previous succession* start-iterationend |

#### 6.7.2.35 Instance: start-iterationend

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startPart |
| owned succession | *owner course* Activity Loop Happening |
| previous succession | *successor* iterationEndPart |

## 6.8 BPMN Extensions

### 6.8.1 Introduction

The BPMN Extension provides additions to the Activity Model for BPMN. These provide BPMN names for special usages of BPDM concepts and additional functionality specific to BPMN. The BPMN Extension includes:

- Activities for scripts, tasks, termination, compensation, and cancelling, along with Embedded processes for transactions.
- Directives for Processes and Embedded Processes, such as adhoc directives.
- Course Control Parts specific to BPMN, such as Inclusive Merge, and specializations of BPDM course control parts, such as Inclusive Decisions.
- User (M1) library for compensation and cancelling.

The descriptions of these and other elements in the BPMN Extension are available in the BPMN specification.

## 6.8.2 Metamodel Specification

The BPMN Extension provides additions to the Activity Model for BPMN. These provide BPMN names for special usages of BPDM concepts and additional functionality specific to BPMN

### 6.8.2.1 Adhoc Extension

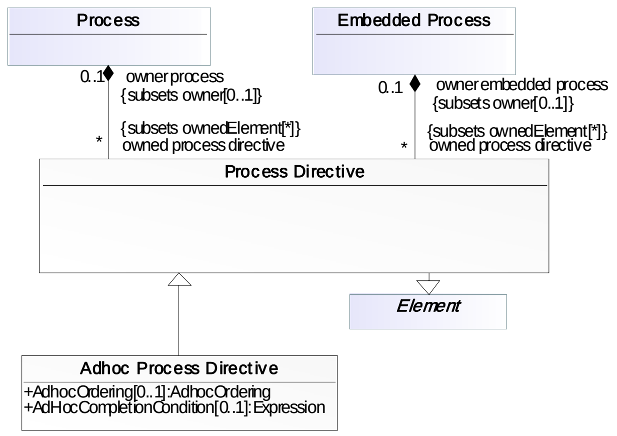


**Figure 6.83 - Adhoc Extension**

### 6.8.2.2 Activity Extensions

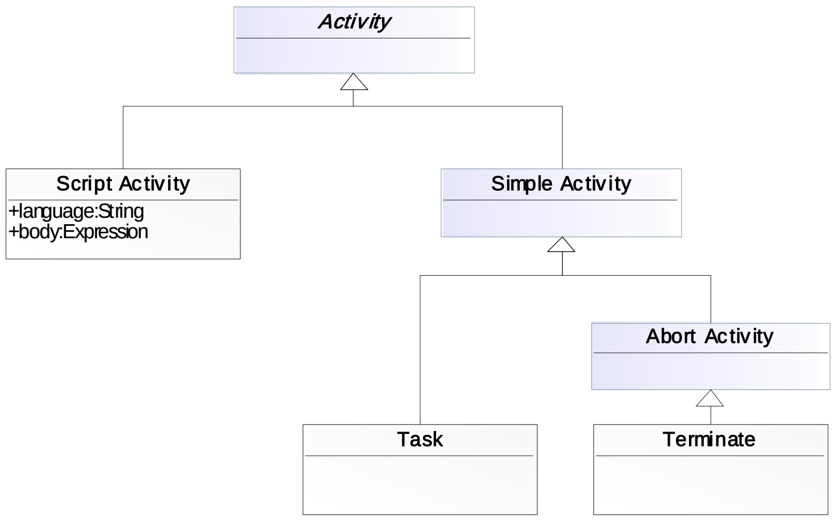


**Figure 6.84 - Activity Extensions**

### 6.8.2.3 Course Control Part Extension

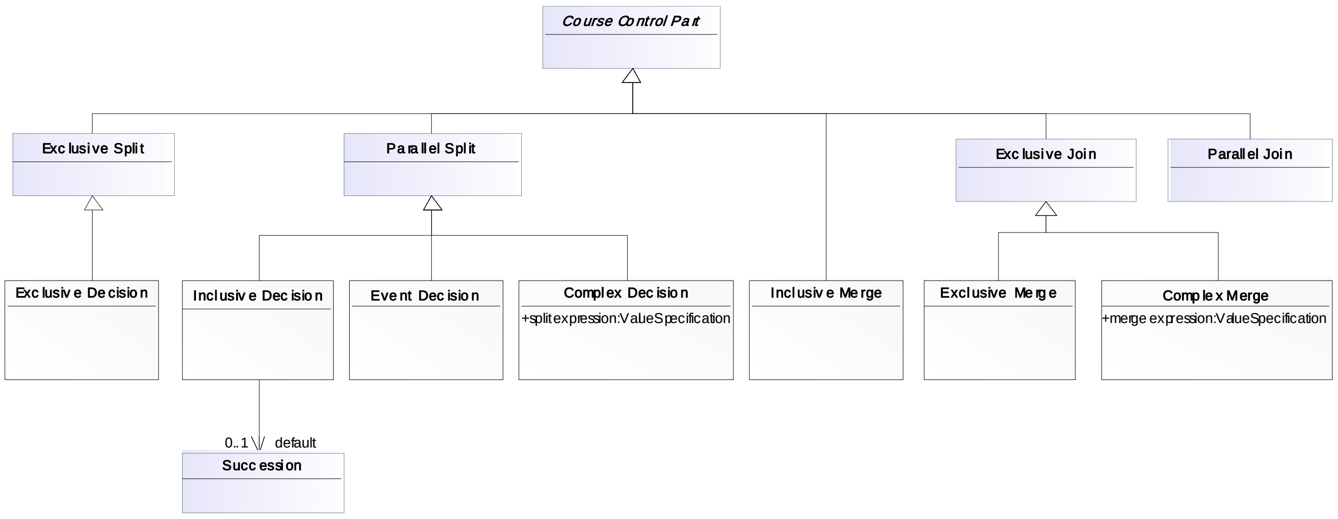
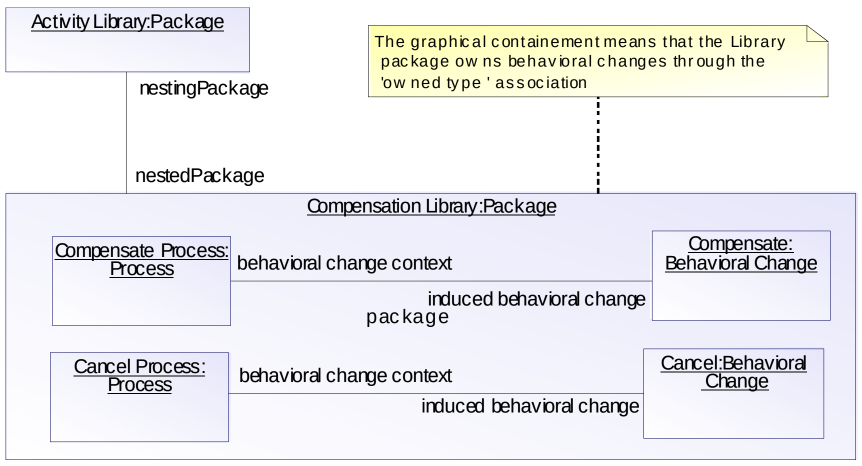


**Figure 6.85 - Course Control Part Extension**

### 6.8.2.4 BPMN Extensions Library: Compensate Process Instance



Activity Library:Package
nestingPackage
nestedPackage
The graphical containement means that the Library package owns behavioral changes through the 'owned type' association
Compensation Library:Package
Compensate Process: Process
behavioral change context
Compensate: Behavioral Change
induced behavioral change
package
Cancel Process: Process
behavioral change context
Cancel:Behavioral Change
induced behavioral change

**Figure 6.86 - BPMN Extensions Library: Compensate Process Instance**

### 6.8.2.5 BPMN Extensions Library: BPMN Universal Process instance

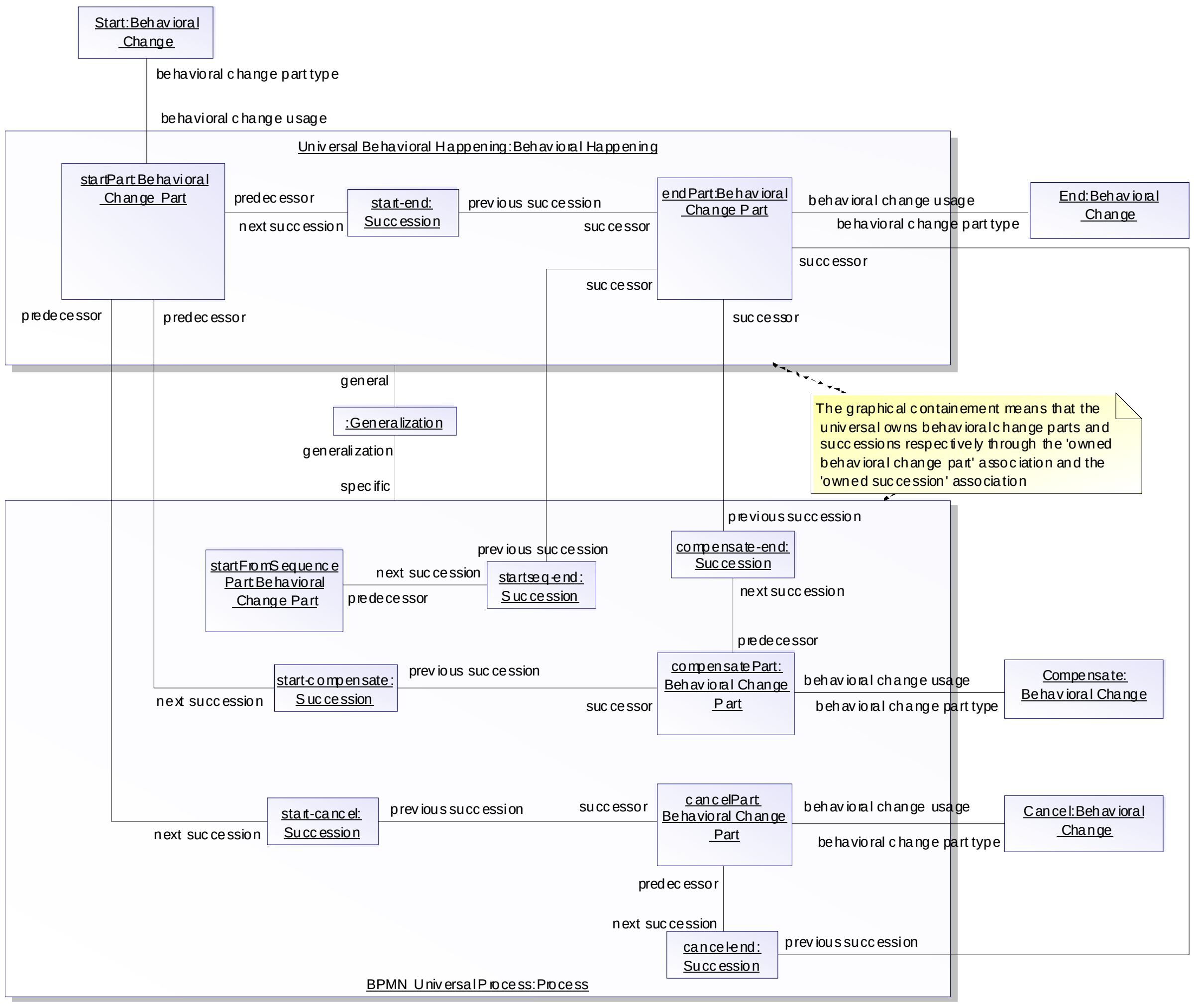

Start:Behavioral Change
behavioral change part type
behavioral change usage
Universal Behavioral Happening:Behavioral Happening
startPart:Behavioral Change Part
predecessor
next succession
start-end: Succession
previous succession
successor
endPart:Behavioral Change Part
behavioral change usage
behavioral change part type
End:Behavioral Change
successor
successor
predecessor
predecessor
successor
general
:Generalization
generalization
specific
The graphical containement means that the universal owns behavioral change parts and successions respectively through the 'owned behavioral change part' association and the 'owned succession' association
previous succession
compensate-end: Succession
previous succession
startFromSequence Part:Behavioral Change Part
next succession
predecessor
startseq-end: Succession
next succession
predecessor
start-compensate: Succession
previous succession
next succession
successor
compensatePart: Behavioral Change Part
behavioral change usage
behavioral change part type
Compensate: Behavioral Change
start-cancel: Succession
previous succession
next succession
successor
cancelPart: Behavioral Change Part
behavioral change usage
behavioral change part type
Cancel:Behavioral Change
predecessor
next succession
cancel-end: Succession
previous succession
BPMN Universal Process:Process

**Figure 6.87 - BPMN Extensions Library: BPMN Universal Process instance**

### 6.8.2.6 Sequence Flow Extension

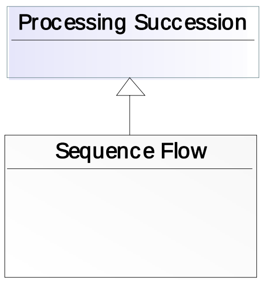


**Figure 6.88 - Sequence Flow Extension**

### 6.8.2.7  Message Extensions

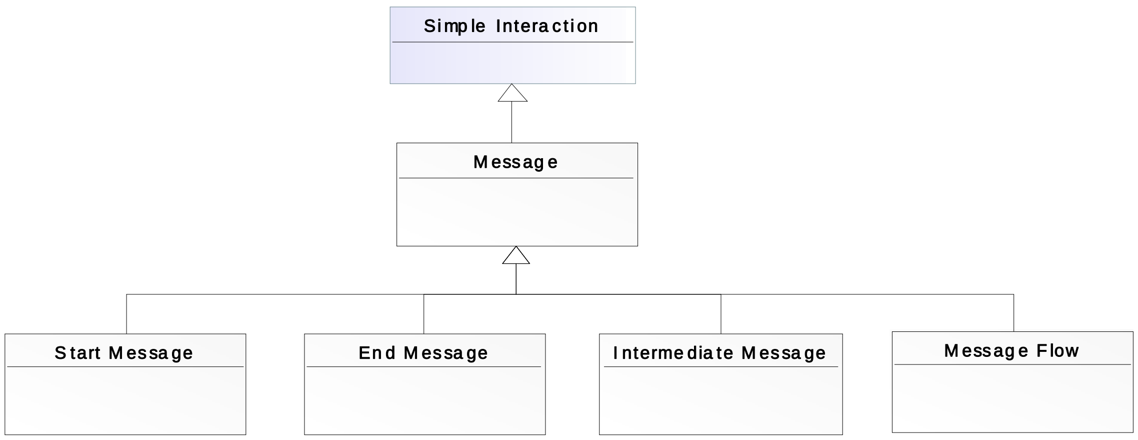


**Figure 6.89 - Message Extensions**

### 6.8.2.8  Artifact Flow Extensions

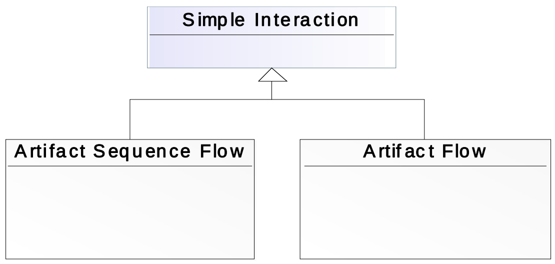


**Figure 6.90 - Artifact Flow Extensions**

### 6.8.2.9  Event Extension

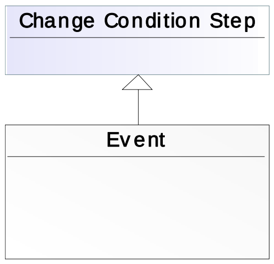


**Figure 6.91 - Event Extension**

### 6.8.2.10  Transaction Extensions

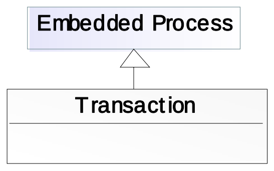


**Figure 6.92 - Transaction Extensions**

### 6.8.2.11 Compensation Extensions

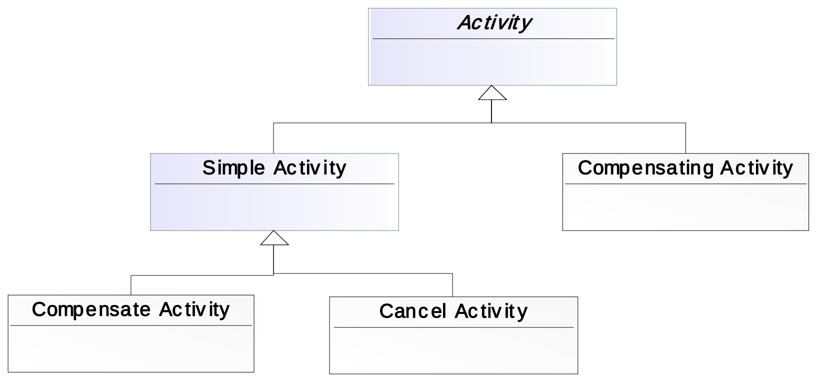


**Figure 6.93 - Compensation Extensions**

### 6.8.2.12 Adhoc Process Directive

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Process Directive”

#### Description

#### Attributes

| | |
|---|---|
| AdhocOrdering: AdhocOrdering [0..1] | |
| AdHocCompletionCondition: Expression [0..1] | |

### 6.8.2.13 AdhocOrdering

**Namespace:** BPMN Extensions
**isAbstract:**

#### Description

Enumeration of the following literal values:

| | |
|---|---|
| parallel: | |

| | |
|---|---|
| sequential: | |

### 6.8.2.14 Artifact Flow

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Simple Interaction”

#### Description

An **Artifact Flow** is a **Simple Interaction** which has a **Holder** as one of its **Interactive Parts**.

### 6.8.2.15 Artifact Sequence Flow

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Simple Interaction”

#### Description

An **Artifact Sequence Flow** is a **Simple Interaction** between two activities that connect an **End Message** from the source **Activity** and a **Start Message** of the target **Activity**.

#### BPMN Notation

An **Artifact Sequence Flow** is represented by a is line with a solid arrowhead that MUST be drawn with a solid single line.

The type of the element transferred by the information flow is represented by a portrait-oriented rectangle that has its upper-right corner folded over that MUST be drawn with a solid single black line.

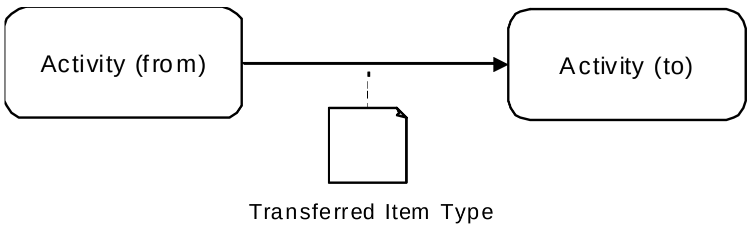


**Figure 6.94 - Artifact Sequence Flow Notation**

### 6.8.2.16 Cancel Activity

**Namespace:** BPMN Extensions
**isAbstract:** No

**Generalization:** “Simple Activity”

### Description

A **Cancel Activity** is a kind of **Simple Activity** that causes the **Cancel** of its enclosing **Behavioral Happening**

In case where the **Cancel Activity** is part of an **Embedded Process**, the cancelled **Behavioral Happening** is this **Embedded Process**, otherwise the cancelled **Behavioral Happening** is the **Process** that owns the **Cancel Activity**

### BPMN Notation

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Cancel** instance of **Behavioral Change**
2. A **Cancel Activity**

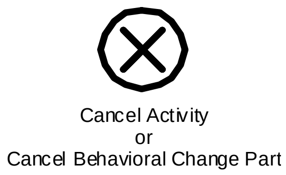


**Figure 6.95 - Cancel Activity Notation or 'Cancel' Behavioral Change Part**

## 6.8.2.17 Compensate Activity

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Simple Activity”

### Description

**Compensate Activity** is a kind of **Simple Activity** that ends a Process and indicates that a Compensation is necessary.

### BPMN Notation

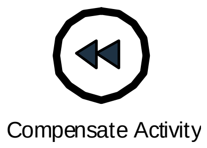


**Figure 6.96 - Compensate Activity Notation**

## 6.8.2.18 Compensating Activity

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Activity”

### Description

A **Compensating Activity** is an **Activity** that follows a **Change Condition Step** conditionned by the **Compensate Behavioral Change**. A **Compensating Activity** cannot have successors.

### Constraint

[1] A **compensating activity**cannot have **next processing succession**

### BPMN Notation

A **Compensating Activity** share the standard activity shape with the **Compensate** marker displayed in the bottom center of the activity.

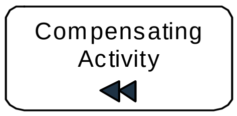


**Figure 6.97 - Compensating Activity Notation**

### 6.8.2.19 Complex Decision

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Parallel Split”

### Description

A **Complex Decision** is a **Parallel Split** that has an expression determining which outgoing **Successions** apply.

### Attributes

| | |
|---|---|
| split expression: ValueSpecification [] | has to evaluate to a boolean value that when evaluated to true enables the split. |

### BPMN Notation

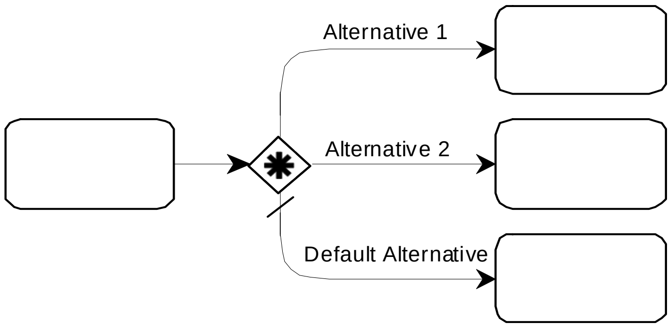


**Figure 6.98 - Complex Decision Notation**

### 6.8.2.20 Complex Merge

**Namespace:** BPMN Extensions

**isAbstract:**

**Generalization:** “Exclusive Join”

#### Description

A **Complex Merge** is an **Exclusive Join** that has an expression determining which which incoming **Successions** must apply for the merge to apply.

#### Attributes

| | |
|---|---|
| merge expression: ValueSpecification [] | |

#### BPMN Notation

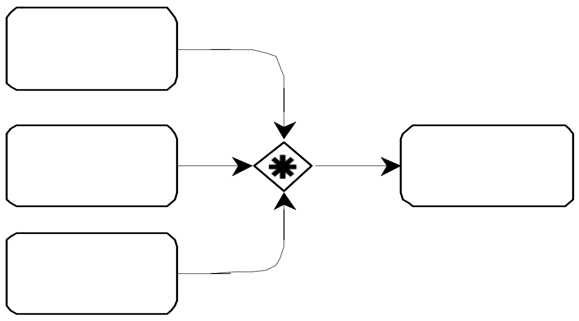


**Figure 6.99 - Complex Join Notation**

### 6.8.2.21 End Message

**Namespace:** BPMN Extensions

**isAbstract:**

**Generalization:** “Message”

#### Description

An **End Message** is a **Message** that has a succession to the **endPart** instance of **Behavioral Change Part** of the **Processing Behavior**. The receipt of this **Message** precedes the end of the **Processing Behavior**.

End Messages are Messages that have a immediate processing succession from their start part to the end part a process. The sending the message is simultaneous with the end of the process.

#### BPMN Notation

Notation for **End Message** or **Simple Interaction** categorized as an **End Message**

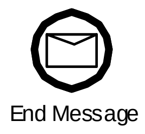


**Figure 6.100 - End Message Notation**

### 6.8.2.22 Event

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Change Condition Step”

#### Description

A BPMN Event is represented by a **Change Condition Step** in BPDM.

### 6.8.2.23 Event Decision

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Parallel Split”

#### Description

An Event Decision applies a race connector to the parts on the target end of processing successions that have the event decision as source (see Processing Behavior). The targeted parts are change condition steps. To wait for incoming messages, these can include behavioral change condition steps detecting the finish of simple interactions from the boundary to the processor role.

#### BPMN Notation

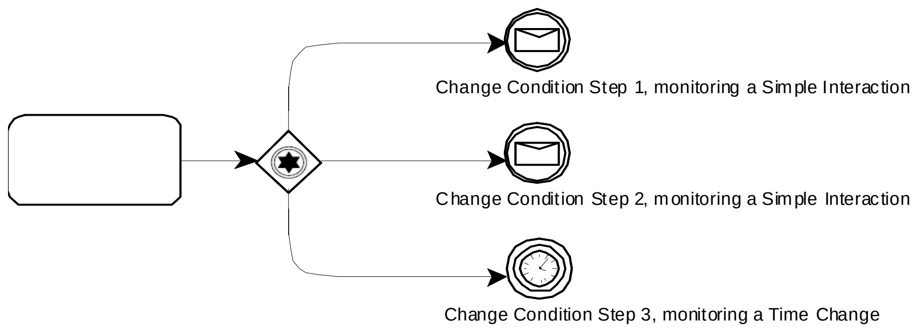

Change Condition Step 1, monitoring a Simple Interaction
Change Condition Step 2, monitoring a Simple Interaction
Change Condition Step 3, monitoring a Time Change

**Figure 6.101 - Event Decision Notation**

### 6.8.2.24 Exclusive Decision

**Namespace:** BPMN Extensions
**isAbstract:**

**Generalization:** “Exclusive Split”

### Description

Same as **Exclusive Split** but with a different name in BPMN

### BPMN Notation

The Exclusive Split shares the same basic shape, called a Gateway,  of the generic **Course Control Part**. The Exclusive Split MAY use a marker that is shaped like an “X” and is placed within the Gateway diamond to distinguish it from other **Course Control Parts**. This marker is not required . A Diagram SHOULD be consistent in the use of the “X” internal indicator. That is, a Diagram SHOULD NOT have some Exclusive Splits with an indicator and some Exclusive Splits without an indicator.

The **default** succession is represented by a default Marker that MUST be a backslash near the beginning of the line representing the **Succession**.

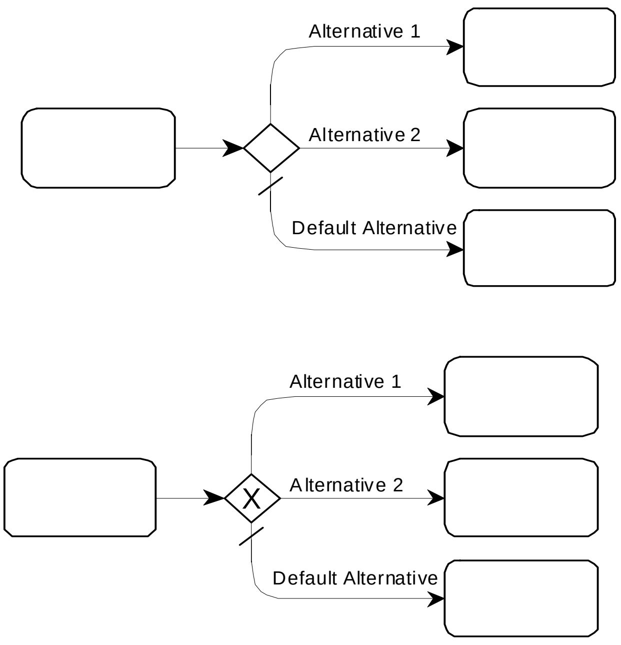


Figure 6.102 - Exclusive Split Notation

## 6.8.2.25  Exclusive Merge

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Exclusive Join”

### Description

Same as **Exclusive Join** but with a different name in BPMN.

### BPMN Notation

The Exclusive Join shares the same basic shape of the generic **Course Control Part**.

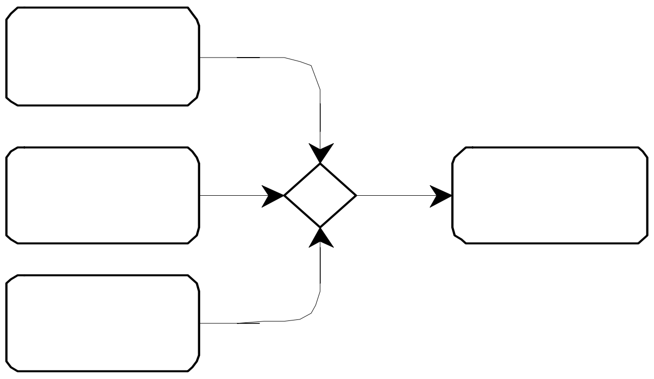


Figure 6.103 - Exclusive Merge Notation

### 6.8.2.26 Inclusive Decision

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Parallel Split”

#### Description

**Inclusive Decision** is a **Parallel Split** that has an outgoing **Succession** specified as the default if none of the other outgoing successions apply due their conditions.

#### Associations

| | |
|---|---|
| default : Succession [0..1] | Succession enabled by default if no other next succession connected to the Inclusive Decision has been enabled. |

### BPMN Notation

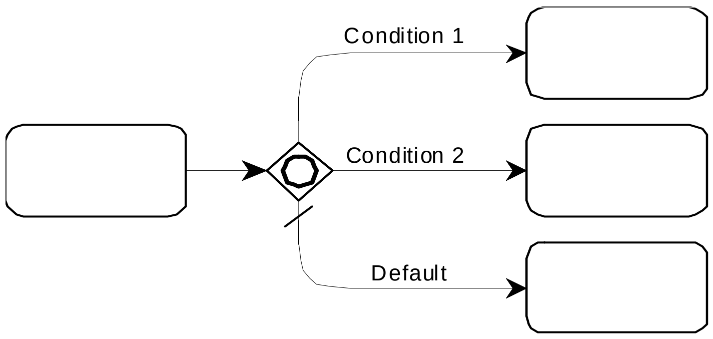


**Figure 6.104 - Inclusive Split Notation**

### 6.8.2.27 Inclusive Merge

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Course Control Part”

### Description

An **Inclusive Merge** is a **Course Control Part** that requires none of the upstream activities to be executing for the join to apply.

### BPMN Notation

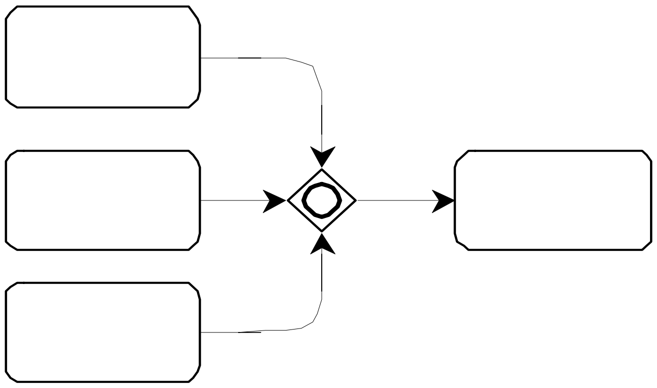


**Figure 6.105 - Inclusive Merge Notation**

### 6.8.2.28 Intermediate Message

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Message”

### Description

An **Intermediate Message** is a **Message** that has a succession to another **Message**.

### BPMN Notation

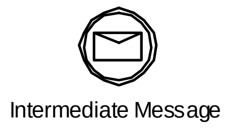


**Figure 6.106 - Intermediate Message Notation or Change Condition Step Monitoring an incoming Message**

## 6.8.2.29 Message

**Namespace:** BPMN Extensions
**isAbstract:** No
**Generalization:** “Simple Interaction”

### Description

A **Message** is a kind of **Simple Interaction** which has an **Interaction Role** as one of its **Interactive Parts**.

### Constraint

[1] At least one of the **Interactive Parts** of a **Message** must be an **Interaction Role**.

### BPMN Notation

The shape of **Message** depends on its sub-types.

The line connecting a **Message** to its **Interaction Role**(s) MUST have an open arrowhead and MUST be drawn with a dashed single black line.

The line connecting a **Message** to other kind of **Interactive Part** MUST have a solid arrowhead and MUST be drawn with a solid single line.

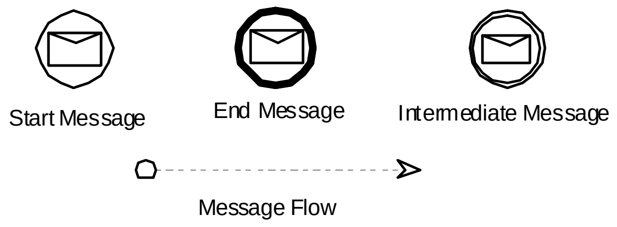


**Figure 6.107 - Message Notation**

## 6.8.2.30 Message Flow

**Namespace:** BPMN Extensions
**isAbstract:**

**Generalization:** “Message”

### Description

An **Message Flow** is a **Message** that has no succession to any other **Message** or **Behavioral Change Part**. Such a **Message** doesn't have any influence on the course of its owning **Processing Behavior**.

### BPMN Notation

A **Message Flow** is line with an open arrowhead that MUST be drawn with a dashed single black line.

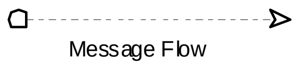


**Figure 6.108 - Message Flow Notation**

## 6.8.2.31 Process Directive

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Element”

### Description

## 6.8.2.32 Script Activity

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Activity”

### Description

### Attributes

| | |
|---|---|
| language: String [] | |
| body: Expression [] | |

## 6.8.2.33 Sequence Flow

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Processing Succession”

### Description

Sequences Flow is Processing Succession from one part to another (see Processing Behavior). If the source part of the succession is typed (not a control part), then if the source part has no intermediate events attached, the source end refers to the end part (which can be omitted as the default), otherwise to the finish part. If the target part is typed, then the target part refers to the start part (which can be omitted as the default).

### BPMN Notation

A Succession is line with a solid arrowhead that MUST be drawn with a solid single line.

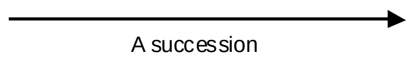


**Figure 6.109 - Succession Notation**

### 6.8.2.34 Start Message

**Namespace:** BPMN Extensions

**isAbstract:**

**Generalization:** “Message”

### Description

A **Start Message** is a **Message** that has a succession to the **startPart** instance of **Behavioral Change Part** of the **Processing Behavior**. The receipt of this **Message** precedes the start of the **Processing Behavior**.

Start Messages are Messages that have a processing succession from their end part to the start part of a process. The receipt of the message creates a new execution of the process.

### BPMN Notation

Notation for **Start Message** or **Simple Interaction** categorized as a **Start Message**.

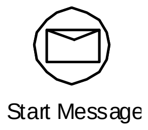


**Figure 6.110 - Start Message Notation**

### 6.8.2.35 Task

**Namespace:** BPMN Extensions

**isAbstract:**

**Generalization:** “Simple Activity”

### Description

BPMN name for **Simple Activity**

### BPMN Notation

An Activity is represented by a rounded corner rectangle that MUST be drawn with a single thin black line.

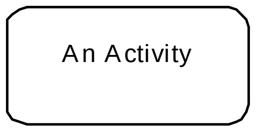


**Figure 6.111 - Activity Notation**

### 6.8.2.36 Terminate

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Abort Activity”

### Description

### BPMN Notation

This symbol can alternatively represent:

1. **Behavioral Change Part** typed by the **Abort** instance of **Behavioral Change**.
2. An **Abort Activity**

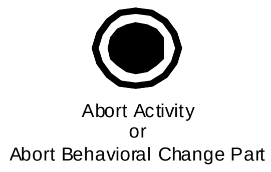


**Figure 6.112 - Abort Activity Notation or 'Abort' Behavioral Change Part**

### 6.8.2.37 Transaction

**Namespace:** BPMN Extensions
**isAbstract:**
**Generalization:** “Embedded Process”

### Description

A **Transaction** is a kind of **Embedded Process** which **enclosed activity** (ies) can be rolled back by the mean of an **Actor**.

### BPMN Notation

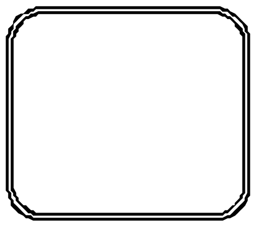


**Figure 6.113 - Transaction Notation**

### 6.8.2.38 Instance: BPMN Universal Process

**Class:** Process

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| | owned succession  startseq-end |
| ownedType | *package*  Compensation Library |
| owner behavioral happening | *owned behavioral change part*  compensatePart |
| owner behavioral happening | *owned behavioral change part*  cancelPart |
| owner course | *owned succession*  start-cancel |
| owner course | *owned succession*  cancel-end |
| owner course | *owned succession*  start-compensate |
| specific | *generalization*  Generalization |

### 6.8.2.39 Instance: Cancel Process

**Class:** Process

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change context | *induced behavioral change*  Cancel |
| ownedType | *package*  Compensation Library |

### 6.8.2.40 Instance: cancel-end

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* cancelPart |
| owned succession | *owner course* BPMN Universal Process |
| previous succession | *successor* endPart |

### 6.8.2.41 Instance: Cancel

**Class:** Behavioral Change

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* cancelPart |
| induced behavioral change | *behavioral change context* Cancel Process |
| ownedType | *package* Compensation Library |

### 6.8.2.42 Instance: cancelPart

**Class:** Behavioral Change Part

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Cancel |
| owned behavioral change part | *owner behavioral happening* BPMN Universal Process |
| predecessor | *next succession* cancel-end |
| successor | *previous succession* start-cancel |

#### 6.8.2.43 Instance: Compensate Process

**Class:** Process

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change context | *induced behavioral change* Compensate |
| ownedType | *package* Compensation Library |

#### 6.8.2.44 Instance: compensate-end

**Class:** Succession

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* compensatePart |
| previous succession | *successor* endPart |

#### 6.8.2.45 Instance: Compensate

**Class:** Behavioral Change

#### Description

**Compensate** is a **Behavioral Change** that manifests that compensation is occuring following an **Abort** of a **Process**

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change part type | *behavioral change usage* compensatePart |
| induced behavioral change | *behavioral change context* Compensate Process |
| ownedType | *package* Compensation Library |

### 6.8.2.46 Instance: compensatePart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| behavioral change usage | *behavioral change part type* Compensate |
| owned behavioral change part | *owner behavioral happening* BPMN Universal Process |
| predecessor | *next succession* compensate-end |
| successor | *previous succession* start-compensate |

### 6.8.2.47 Instance: Compensation Library

**Class:** Package

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| nestedPackage | *nestingPackage* Activity Library |
| package | *ownedType* Cancel |
| package | *ownedType* Compensate |
| package | *ownedType* BPMN Universal Process |
| package | *ownedType* Cancel Process |
| package | *ownedType* Compensate Process |

### 6.8.2.48 Instance: Generalization

**Class:** Generalization

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|

|  |  |
|---|---|
|  | general  Universal Behavioral Happening |
| generalization | *specific*  BPMN Universal Process |

### 6.8.2.49  Instance: start-cancel

**Class:** Succession

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor*  startPart |
| owned succession | *owner course*  BPMN Universal Process |
| previous succession | *successor*  cancelPart |

### 6.8.2.50  Instance: start-compensate

**Class:** Succession

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor*  startPart |
| owned succession | *owner course*  BPMN Universal Process |
| previous succession | *successor*  compensatePart |

### 6.8.2.51  Instance: startFromSequencePart

**Class:** Behavioral Change Part

#### Description

#### Links

| Played End | *Opposite End* |
|---|---|
| predecessor | *next succession*  startseq-end |

#### 6.8.2.52 Instance: startseq-end

**Class:** Succession

**Description**

**Links**

| Played End | *Opposite End* |
|---|---|
| next succession | *predecessor* startFromSequencePart |
| owned succession | *BPMN Universal Process* |
| previous succession | *successor* endPart |

## 6.9 Interaction Protocol Model

### 6.9.1 Introduction

The Interaction Protocol Model is for capturing choreographies. It enables interactions to be grouped together into larger, reusable interactions. For example, an interaction that exchanges goods between companies might be used with other interactions within a larger protocol representing a partnership of the companies. This protocol might be adopted by a standards body and reused between many pairs of companies. The interactions in a protocol may be simple interactions that have no sub-interactions, or may be other protocols.

The Interaction Model provides:

- Processing behaviors with steps that are interactions (Interaction Protocols).
- Interactions representing the reuse of protocols (Compound Interactions).
- A way to specify how a reused protocol ties in with the protocols using it (Interactive Part Binding).

Interaction Protocols are Processing Behaviors where the Behavioral Steps are Interactions. For example, a protocol between two companies might start with one company sending another an order, then the other sending back a product, and then the original company sending payment, and finally receiving a receipt. These four simple interactions can be grouped into an interaction protocol, with processing successions between them to specify which interaction comes before which (see the Processing Behavior Model). The two companies are interaction roles in the protocol (see the Simple Interaction Model).

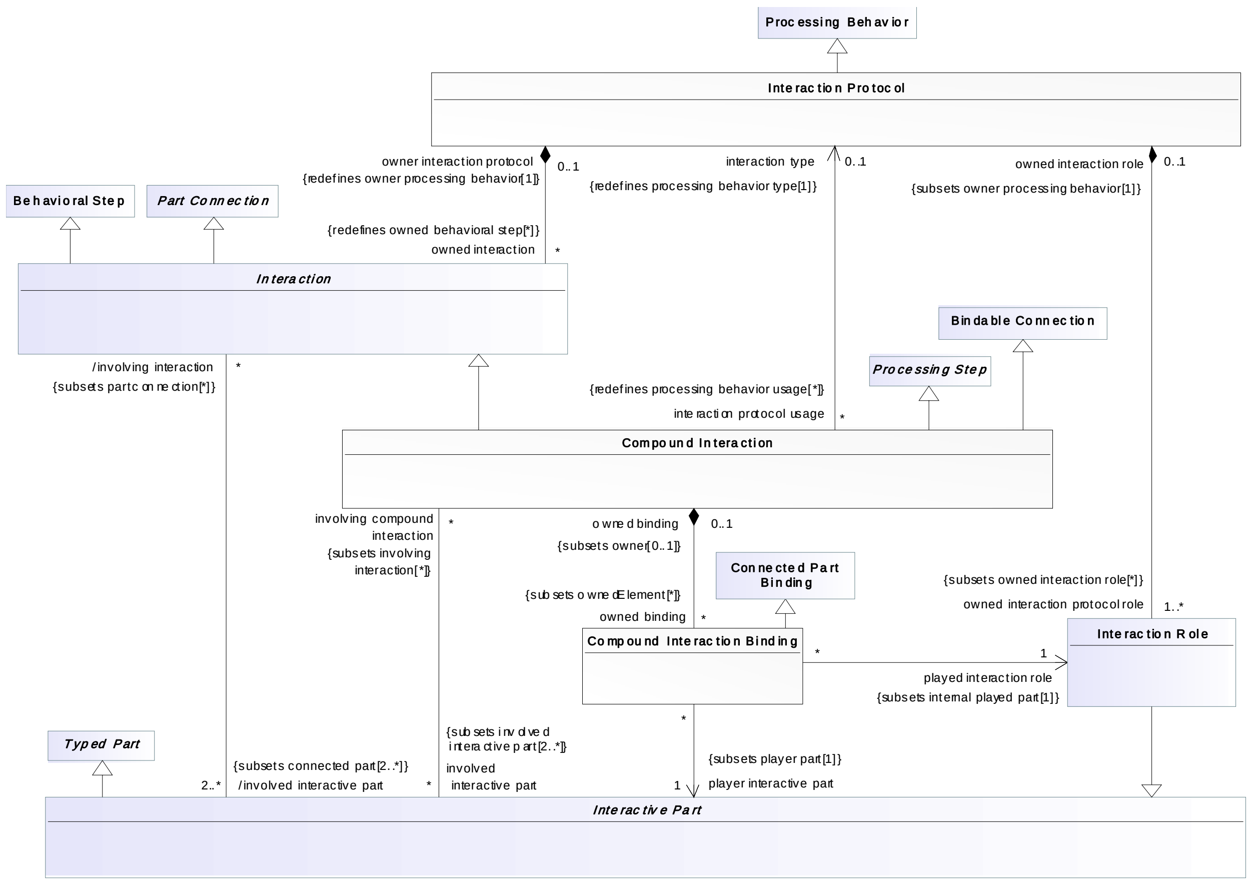
Compound Interactions are Interactions that are also Processing Steps, enabling them to reuse Interaction Protocols. For example, two companies might use the ordering protocol described above many times as part of a larger partnership. This partnership is a larger interaction protocol that reuses the ordering protocol many times. Each reuse is represented as a compound interaction in the larger partnership protocol. Compound Interactions are complementary to Simple Interactions, which are Interactions that do not have sub-interactions (see the Simple Interaction Model).

Interactive Part Bindings are Part Bindings that specify how a protocol reused by a Compound Interaction ties in with the larger protocol reusing it (see the Part Binding subsection in the Composition Model). For example, reusing the ordering protocol described above requires specifying which part in the larger partnership identifies the buying company and which identifies the selling company. Both companies will play these roles at some point in the larger partnership, so the bindings must be specified for each compound interaction.

### 6.9.2 Metamodel Specification

The Interaction Protocol Model is for capturing choreographies. It enables interactions to be grouped together into larger, reusable interactions. For example, an interaction that exchanges goods between companies might be used with other interactions within a larger protocol representing a partnership of the companies. This protocol might be adopted by a standards body and reused between many pairs of companies. The interactions in a protocol may be simple interactions that have no sub-interactions, or may be other protocols.

#### 6.9.2.1 Interaction Protocol



Processing Behavior
Interaction Protocol
owner interaction protocol
{redefines owner processing behavior[1]}
0..1
interaction type
{redefines processing behavior type[1]}
0..1
owned interaction role
{subsets owner processing behavior[1]}
0..1
Behavioral Step
Part Connection
{redefines owned behavioral step[*]}
owned interaction
*
Interaction
Bindable Connection
Processing Step
/involving interaction
{subsets partconnection[*]}
*
{redefines processing behavior usage[*]}
interaction protocol usage
*
Compound Interaction
involving compound interaction
{subsets involving interaction[*]}
*
owned binding
{subsets owner[0..1]}
0..1
Connected Part Binding
{subsets ownedElement[*]}
owned binding
*
{subsets owned interaction role[*]}
owned interaction protocol role
1..*
Compound Interaction Binding
*
1
Interaction Role
played interaction role
{subsets internal played part[1]}
*
Typed Part
{subsets connected part[2..*]}
2..*
/involved interactive part
{subsets involved interactive part[2..*]}
involved interactive part
*
{subsets player part[1]}
player interactive part
1
Interactive Part

**Figure 6.114 - Interaction Protocol**

#### 6.9.2.2 Compound Interaction

**Namespace:** Interaction Protocol Model

**isAbstract:** No

**Generalization:** “Bindable Connection” “Interaction” “Processing Step”

### Description

A **Compound Interaction** is an **Interaction** that is also a **Processing Step**, enabling it to reuse an **Interaction Protocol**. **Compound Interaction** is complementary to **Simple Interaction**, which is an **Interaction** that doesn't have sub-interactions.

### Associations

| | |
|---|---|
| interaction type : Interaction Protocol [0..1] | Interaction Protocol that defines the type of the Compound Interaction<br>Redefines *processing behavior type* |

| | |
|---|---|
| involved interactive part : Interactive Part [*] | Interactive Part involved in the Compound Interaction<br>Subsets *involved interactive part* |

| | |
|---|---|
| owned binding : Compound Interaction Binding [*] | Subsets *ownedElement* |

### Non Normative Notation

A compound interaction is represented by a rounded corner rectangle that MUST be drawn with a double thin black line.

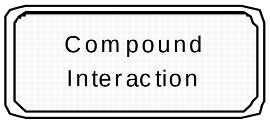


**Figure 6.115 - Compound Interaction Notation**

## 6.9.2.3 Compound Interaction Binding

**Namespace:** Interaction Protocol Model

**isAbstract:** No

**Generalization:** “Connected Part Binding”

### Description

An **Compound Interaction Binding** is a **Connected Part Binding** that specifies how an **Interaction Protocol** reused by a **Compound Interaction** ties in with the larger **Processing Behavior** reusing it.

For each **Interactive Part** involved in a **Compound Interaction**, there is a **Compound Interaction Binding** that specifies which **Interaction Role** it plays in the **Interaction Protocol**.

**Associations**

| | |
|---|---|
| played interaction role : Interaction Role [1] | The Interaction Role that is played by the player interactive part connected by the Compound Interaction Binding<br>Subsets *internal played part* |

| | |
|---|---|
| player interactive part : Interactive Part [1] | The Interactive PartInteractive Part being playing the played interaction role as defined by the Compound Interaction Binding<br>Subsets *player part* |

### 6.9.2.4 Interaction Protocol

**Namespace:** Interaction Protocol Model
**isAbstract:** No
**Generalization:** “Processing Behavior”

**Description**

An **Interaction Protocol** is a kind of **Processing Behavior** where **Behavioral Steps** are **Interactions** that occur between **Interaction Roles**.

The set of **Interaction**s defines the purpose of the **Interaction Protocol**.

**Associations**

| | |
|---|---|
| owned interaction protocol role : Interaction Role [1..*] | Interaction Role owned by the Interaction Protocol<br>Subsets *owned interaction role* |

| | |
|---|---|
| owned interaction : Interaction [*] | Redefines *owned behavioral step* |

## 6.10 Class Hierarchies

The **Class Hierarchies** is not a real package. It groups diagrams that provide a synthesis of class hierarchies.

### 6.10.1 Happening OverTime Hierarchy

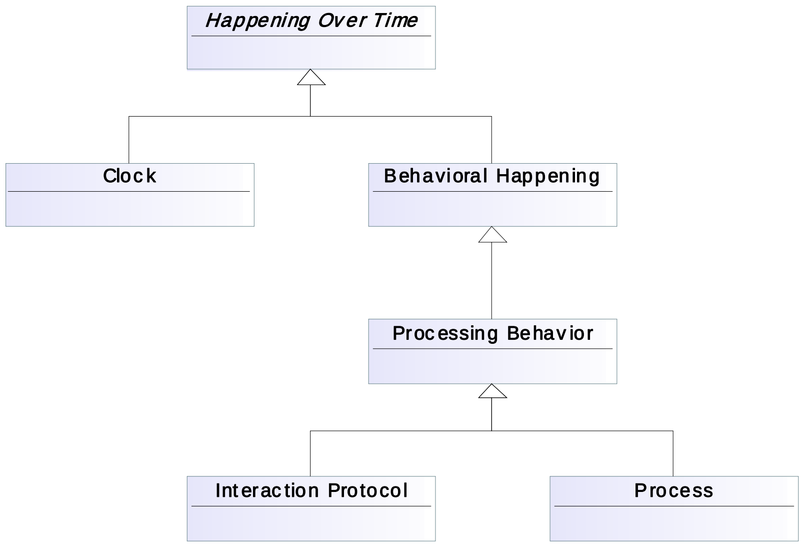


**Figure 6.116 - Happening OverTime Hierarchy**

### 6.10.2 Change Hierarchy

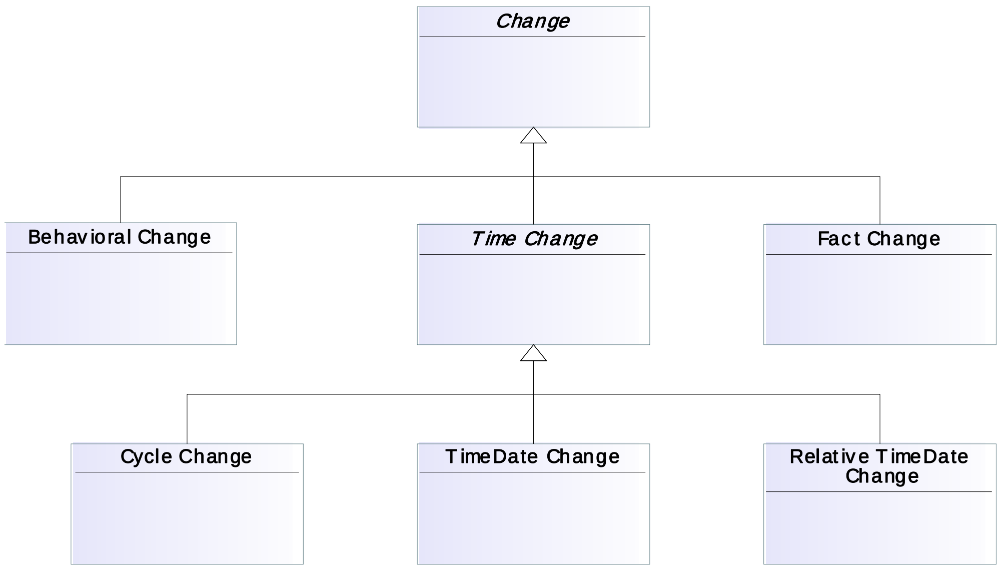


**Figure 6.117 - Change Hierarchy**

### 6.10.3 Succession Hierarchy

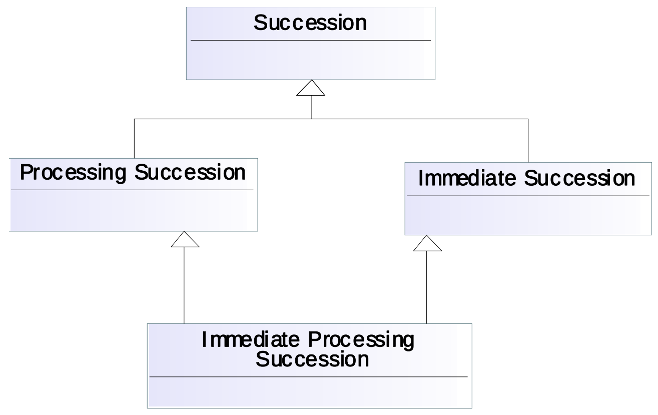


**Figure 6.118 - Succession Hierarchy**

### 6.10.4 Behavioral Step Hierarchy

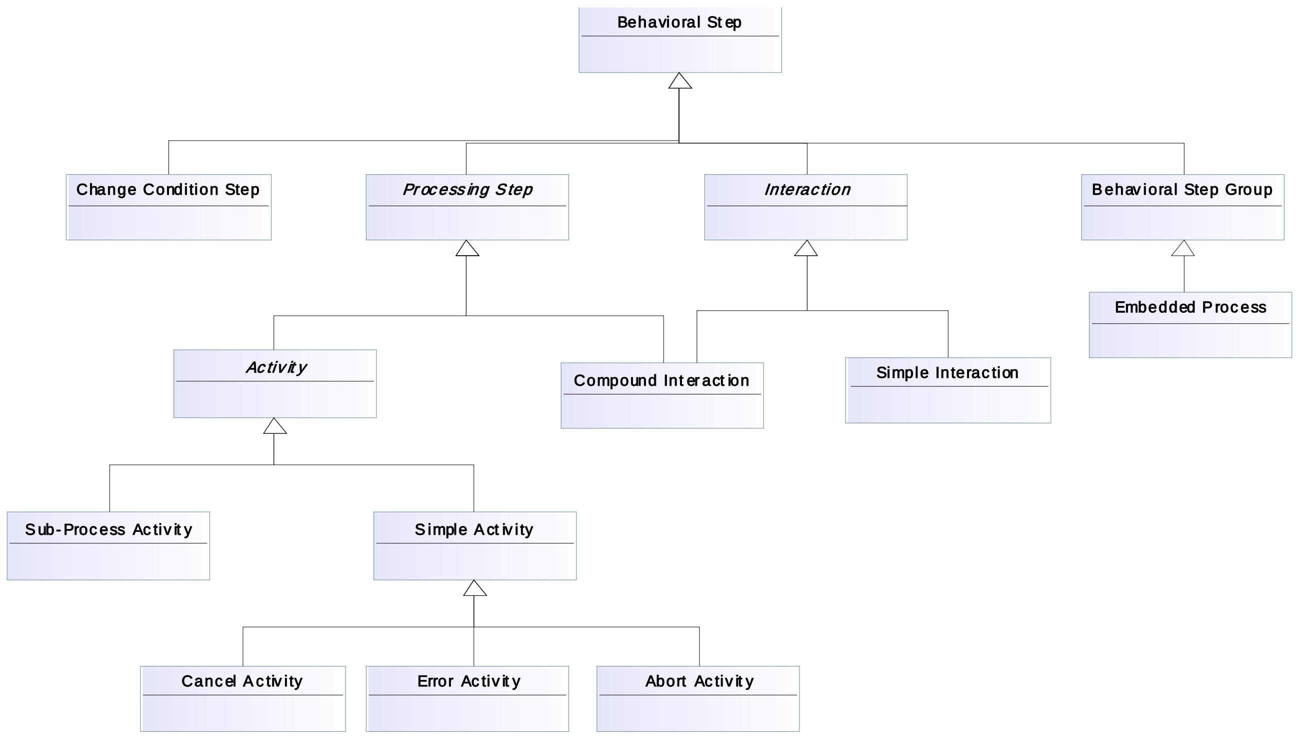


**Figure 6.119 - Behavioral Step Hierarchy**

### 6.10.5 Interactive Part Hierarchy

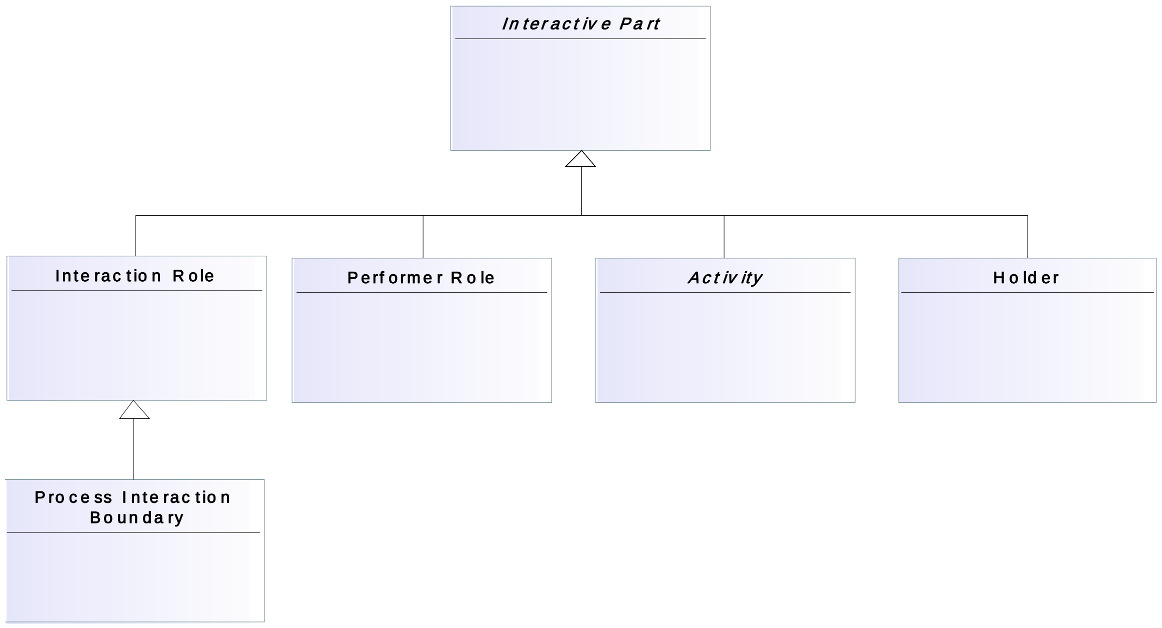


**Figure 6.120 - Interactive Part Hierarchy**

### 6.10.6 Simple Interaction Hierarchy

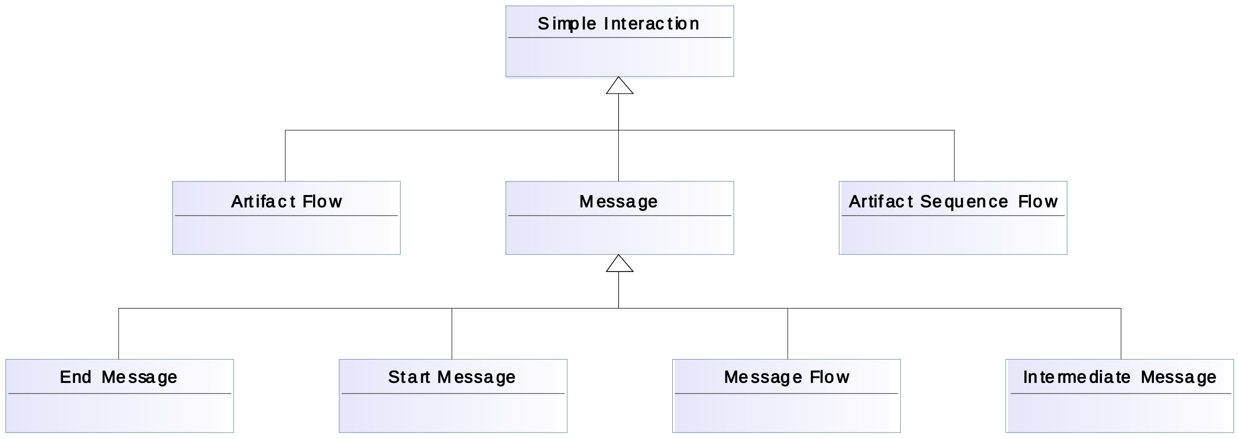


**Figure 6.121 - Simple Interaction Hierarchy**

# 7 BPDM-BPEL Mapping

## 7.1 General

This document will cover a non-normative mapping from BPDM constructs to WS-BPEL 2.0 elements. The basis for the mapping are the “Mapping to BPEL” in [BPMN] (Section 11) and “BPMN to BPDM Mapping” in [BPDM] (Section 6).

### 7.1.1 Topological Considerations

The Business Process Definition Metamodel (BPDM) is a graph-oriented language in which control and action nodes can be connected almost arbitrarily. In contrast, Business Process Execution Language (BPEL) is a mainly block-structured (albeit providing graph-oriented constructs with syntactical limitations) language with a properly nested structure. As BPDM and BPEL represent two fundamentally different classes of languages, the mapping is technically challenging; while BPEL to BPDM mapping is trivial, not all BPDM processes can easily be converted to BPEL.

To map a BPDM process onto BPEL code, a transformation from a graph structure to block structure is needed. For this purpose, the process can be decomposed into components with one entry and one exit point [BPM-06-02]. These components can then be mapped onto suitable “BPEL blocks.” The decomposition helps to define an iterative approach which allows an incremental transformation of a “componentized” BPDM process to a block-structured BPEL process.

A component may be well-structured so that it can be directly mapped onto BPEL structured activities, whereas a non-well-structured component can be translated into BPEL via event-action rules. The latter approach can be applied to translating any component to BPEL, yet it produces less readable BPEL code and will therefore be applied only to the remaining non-well-structured components. The algorithm is explained in detail in [BPM-06-02] that addresses the same problem in translation between BPMN and BPEL.

### 7.1.2 Generator Model

In general transformation from one metamodel to another metamodel requires additional information. This information is provided in a separate model that is specific to the performed transformation. We will refer to this model as “generator model.”

If information required by BPEL and not provided by BPDM is needed then the generator model is responsible for providing it. Such examples are: XML namespaces, specific BPEL customizations, etc. Using the generator model we could avoid introducing concepts and terms in BPDM that are specific for BPEL and still have the capability to customize the produced BPEL models.

Ultimately, a generator metamodel would be required for this generator model in order to describe all possible customizations that can be used. For the purposes of this non-normative mapping, however, it is merely indicated which additional information is needed for the mapping (see Notational Conventions).

### 7.1.3 Notational Conventions

BPDM constructs are depicted in **Bold** typeface. The equivalent BPMN element may follow in (Parentheses). BPEL elements are represented in <angle brackets> and attributes in italics. Marks are denoted in ***Bold Italics***.

The keywords “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

## 7.2 Process

| BPDM | BPEL |
|---|---|
| Processor Role | **Processor Role** maps to BPEL &lt;process&gt; element. The **NamedElement.name** maps to the name attribute of &lt;process&gt;. |

## 7.3 Start Event Mappings

| BPDM | BPEL |
|---|---|
| Behavioral Change Part typed by the Start Behavioral Change | The only way to instantiate a business process in BPEL is to annotate a &lt;receive&gt; or &lt;pick&gt; activity with the createInstance attribute set to “yes.” The &lt;receive&gt; or &lt;pick&gt; will likely be placed within a &lt;sequence&gt; or a &lt;flow&gt;. |
| Start Message | This will map to the &lt;receive&gt; element. The createInstance attribute of the &lt;receive&gt; element will be set to “yes.”<br>The **Message** attribute of **Start** maps to the variable attribute of the &lt;receive&gt; element. Note that the extra spaces and non-alphanumeric characters MUST be stripped from the variable attribute to fit with the XML specification of the variable attribute. If there is a name collision (because of the name change), then the transformer is responsible for generating unique names.<br>The **Name** attribute of **Simple Interaction** maps to the name attribute of a BPEL &lt;variable&gt; element. Note that the extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the name attribute. Note that there may be two or more elements with the same name after **Name** has been stripped.<br>The messageType, type or element attribute is used to specify the type of a variable. Exactly one of these attributes MUST be used. The messageType attribute of the variable element refers to a WSDL message type definition. Thus, the messageType will share the same Name and a corresponding WSDL message must be created. Attribute type refers to an XML Schema type (simple or complex). Attribute element refers to an XML Schema element. |

| | |
|---|---|
| Start Message | In case of using a WSDL message type definition, each **Properties** will map to a <part> element of the WSDL <message>. The **Name** attribute of the **Property** will map to the name attribute of the <part>. The **Type** attribute of the **Property** will map to the type attribute of the <part>.<br><br>The **Implementation** attribute of **Simple Interaction** MUST be a Web service or MUST be converted to a Web service for mapping to BPEL. The Web Service Attributes are mapped as follows:<br>• The ***Participant*** attribute is mapped to the *partnerLink* attribute of the BPEL activity<br>• The ***Interface*** attribute is mapped to the *portType* attribute of the BPEL activity<br>• The ***Operation*** attribute is mapped to the *operation* attribute of the BPEL activity<br><br>**InteractionFlow.transformationExpression** will map to a <fromParts> element within <receive>. |
| **Time Condition** on **Start** | This will map to the <receive> element. The createInstance attribute of the <receive> element will be set to "yes."<br><br>The remaining attributes of the <receive> will be mapped as as shown for the Message Start Event (see above).<br><br>During the mapping an additional BPEL process is employed. We will refer to this process as <NameOfStartNode>trigger. Thus the functionality of the timing as defined in the Start Event will be implemented in a separate process that will be started by the BPEL Engine. The process definition will use a <wait> element for the defined time, and then use an <invoke> to send a message that will be received by the above <receive> element. A specific Message and Web service implementation must be provided so that the mappings to <receive> element can be completed.<br><br>**InteractionFlow.transformationExpression** will map to a <fromParts> element within <receive>. |
| **Fact Change Condition** on **Start** | This will map to the <receive> element. The createInstance attribute of the <receive> element will be set to "yes."<br><br>The remaining attributes of the <receive> element will be mapped as shown for the Message Start Event (see above).<br><br>**InteractionFlow.transformationExpression** will map to a <fromParts> element within <receive>.<br><br>Note: the Message is expected to arrive from the application that tracks and triggers Business Rules. |

## 7.4 End Event Mappings

| **BPDM** | **BPEL** |
|---|---|
| End Behavioral Change Part | There is no BPEL element that **Finish** will map to. However, it marks the end of a path within the Process and will be used to define the boundaries of complex BPEL elements. |
| End Message | This will map to a BPEL <reply> or an <invoke>. The appropriate BPEL activity will be determined by the implementation defined for the Event. That is, the portType and operation of the Message will be used to check to see if an upstream Message Event has the same portType and operation. If these two attributes are matched, then the Event will map to a <reply>, if not, the Event will map to an <invoke>.<br>The **Message** attribute of **Finish** maps to the variable attribute of the <reply> or the outputVariable of the <invoke>.<br>See the corresponding Message Start Event above for more information about how **Simple Interaction** maps to BPEL and WSDL.<br>The **Implementation** attribute of **Simple Interaction** MUST be a Web service or MUST be converted to a Web service for mapping to BPEL. The Web Service Attributes are mapped as follows:<br>• The **Participant** attribute is mapped to the partnerLink attribute of the BPEL activity<br>• The **Interface** attribute is mapped to the portType attribute of the BPEL activity<br>• The **Operation** attribute is mapped to the operation attribute of the BPEL activity<br>**InteractionFlow.transformationExpression** will map to the fromVariable variable of <toParts> element within <reply> or <invoke>. |
| Error Activity | This will map to a <throw> element. The **ErrorCode** attribute of **Error Activity** will map to the faultName attribute of the <throw>. |
| Cancel Activity | The mapping of Cancel Activity to BPEL is an open issue. |
| Abort Activity | This will map to the <exit> element. |

## 7.5 Intermediate Events

| BPDM | BPEL |
|---|---|
| **Simple Interaction** coming from or going to the **Process Interaction Boundary** that is not connected to **Start** or **Finish**. | If **Simple Interaction.Simple Interaction consumer** refers to the same Participant as that of the Process that contains the Event, then this will map to a <receive>. The createInstance attribute of the <receive> element will be set to "no." If **Simple Interaction.Simple Interaction producer** is the same Participant as that of the Process that contains the Event, then this will map to a (one-way) <invoke>.<br>The **Message** attribute of the Event maps to the variable attribute of the <receive> or the outputVariable of the <invoke>.<br>See the corresponding Start event above for more information about how **Simple Interaction** maps to BPEL and WSDL. |
| | The **Implementation** attribute of **Simple Interaction** MUST be a Web service or MUST be converted to a Web service for mapping to BPEL. The Web Service Attributes are mapped as follows:<br>• The **Participant** attribute is mapped to the partnerLink attribute of the BPEL activity<br>• The **Interface** attribute is mapped to the portType attribute of the BPEL activity<br>• The **Operation** attribute is mapped to the operation attribute of the BPEL activity<br>If the Event has no incoming **Processing Succession**:<br>• **Simple Interaction.Simple Interaction consumer** MUST be the same Participant as that of the Process that contains the Event.<br>• The <process> could be given a <scope> (if it doesn't already have one).<br>• An <eventHandlers> element can be defined directly under <process> or under <scope> (if one was generated).<br>• An <onMessage> element will be added to the <eventHandlers> element.<br>• The **Message** attribute of the Event maps to the variable attribute of the <onMessage>.<br>Further, The **Implementation** attribute of **Simple Interaction** MUST be a Web service or MUST be converted to a Web service for mapping to BPEL. The Web Service Attributes are mapped as follows:<br>• The **Participant** attribute is mapped to the partnerLink attribute of the <onMessage><br>• The **Interface** attribute is mapped to the portType attribute of the <onMessage><br>• The **Operation** attribute is mapped to the operation attribute of <onMessage> |

| | |
|---|---|
| Processing Succession from the abort Change Part | The mappings of the activity (to which the Event is attached) will be placed within a <scope>.<br>A <faultHandlers> element will be defined for the scope.<br>A <catch> element will be added to the <faultHandlers> element with "<message name>_Exit" as the faultName attribute.<br>An <eventHandlers> element will be defined for the scope.<br>The Event will map to an <onMessage> element within the <eventHandlers>. The mapping to the <onMessage> attributes is the same as described for the <receive> above.<br>The activity for the <onMessage> will be a <throw> with "<message name>_Exit" as the faultName attribute.<br>If used in an event-based decision, this will map to an <onMessage> within a <pick>. The mapping to the <onMessage> attributes is the same as described for the <receive> above. |
| **Time Change Condition** on **Succession** | This will map to a <wait>.<br>**TimeChange.timeExpression** maps to the until attribute of the <wait>.<br>**Cycle Change.timeExpression** maps to the for attribute of the <wait>.<br>If the Event has no incoming **Processing Succession**:<br>• The <process> could be given a <scope> (if it doesn't already have one).<br>• An <eventHandlers> element will be defined for the process or the <scope> (if <scope> element was generated).<br>• An <onAlarm> element will be added to the <eventHandlers> element.<br>• **TimeChange.timeExpression** maps to the until attribute of the <onAlarm>.<br>• **Cycle Change.timeExpression** maps to the for attribute of the <onAlarm>. |
| **Racing Connection** connecting a **Change Condition Step** conditioned by a **Time Change Condition** | The mappings of the activity (to which the Event is attached) will be placed within a <scope>.<br>A <faultHandlers> element will be defined for the scope.<br>A <catch> element will be added to the <faultHandlers> element with "<Event name>_Exit" as the faultName attribute.<br>An <eventHandlers> element will be defined for the scope.<br>The Event will map to an <onAlarm> element within the <eventHandlers>.<br>**TimeChange.timeExpression** maps to the until attribute of the <onAlarm>.<br>**Cycle Change.timeExpression** maps to the for attribute of the <onAlarm>.<br>The activity for the <onAlarm> will be a <throw> with "<message name>_Exit" as the faultName attribute.<br>If used in an event-based decision, this will map to an <onAlarm> within a <pick>.<br>**TimeChange.timeExpression** then maps to the until attribute of the <onAlarm>.<br>Accordingly, **Cycle Change.timeExpression** maps to the for attribute of the <onAlarm>. |

| | |
|---|---|
| Processing Succession from the errorPart | Within the normal flow, **Processing Succession** will map to a <throw> element.<br>If the error is attached to an activity, the mappings of the activity (to which the Event is attached) will be placed within a <scope>.<br>This Event will map to a <catch> element within a <scope>.<br>If the Error Behavioral Change does not have an **ErrorCode**, then a <catchAll> element will be added to the <faultHandlers> element.<br>If the Error Behavioral Change has an **ErrorCode**, then a <catch> element will be added to the <faultHandlers> element with the **ErrorCode** mapping to the faultName attribute. |
| Processing Succession from the abortPart | The mapping of succession from abort to BPEL is an open issue. |
| **Fact Change Condition** on **Succession** | This will map to the <receive> element. The createInstance attribute of the <receive> element will be set to “no.” The remaining attributes of the <receive> will be mapped as shown for the Message Start Event (see above).<br>If the Event has no incoming **Processing Succession**:<br>• Simple Interaction.Simple Interaction consumer MUST be the same Participant as that of the Process that contains the Event.<br>• The <process> could be given a <scope> (if it doesn’t already have one).<br>• An <eventHandlers> element will be defined for the process or the <scope> (if one was generated).<br>• The Event will map to an <onMessage> element within the <eventHandlers>. The mapping to the <onMessage> attributes is the same as described for the <receive> for the Message Event above.<br>Note: the Message is expected to arrive from the application that tracks and triggers |
| **Racing Connection** connecting **a Change Condition Step** monitoring **a Fact Change Condition** | The mappings of the activity (to which the Event is attached) will be placed within a <scope>.<br>A <faultHandlers> element will be defined for the scope.<br>A <catch> element will be added to the <faultHandlers> element with “<message name>_Exit” as the faultName attribute.<br>An <eventHandlers> element will be defined for the scope.<br>The Event will map to an <onMessage> element within the <eventHandlers>. The mapping to the <onMessage> attributes is the same as described for the <receive> for the Message Event above.<br>Note: the Message is expected to arrive from the application that tracks and triggers Business Rules.<br>The activity for the onMessage will be a <throw> with “<message name>_Exit” as the faultName attribute.<br>If used in an event-based decision, this will map to an <onMessage> element within <pick>. The mapping to the <onMessage> attributes is the same as described for the <receive> for the Message Event above. |

| | |
|---|---|
| **Change Condition Step** monitoring a **Compensation Change** | Within the normal flow:<br>Maps to a <compensate> or <compensateScope> element. The **Name** of the activity referenced by the Compensation Event will map to the target attribute of the <compensateScope> element.<br>Attached to an activity boundary:<br>The activity (to which the Event is attached) will be placed within a <scope>. This Event maps to a <compensationHandler> element within a <scope>.<br>For the <invoke> activity, there is a special shortcut to inline a <compensationHandler> within <invoke> rather than explicitly using an immediately enclosing scope. |

## 7.6 Activities

| **BPDM** | **BPEL** |
|---|---|
| Simple Activity | An incoming **Simple Interaction** maps to a <receive> activity. The **Message** attribute maps to the variable attribute of the <receive> activity. If the **Simple Interaction** represents start Simple Interaction, then the createInstance attribute of the receive will be set to "yes."<br>Two **Simple Interaction**s associated with the same activity – an incoming and an outgoing flow – map to an <invoke> activity. The **InMessage** attribute maps to the inputVariable attribute of the <invoke> activity. The **OutMessage** attribute maps to the outputVariable attribute.<br>An outgoing **Simple Interaction** maps to a <reply> or an <invoke> activity. The appropriate BPEL activity will be determined by checking if an upstream <receive> has the same portType and operation. If these two attributes are matched, then the activity will map to a <reply>, if not, it will map to an <invoke>. The **Message** attribute maps to the variable attribute of the <reply> activity or it maps to the inputVariable attribute of the <invoke> activity.<br>See the Start event above for more information about how **Simple Interaction** maps to BPEL and WSDL. |
| Script Activity | This maps to an <invoke> activity. Since this activity is performed by a process engine, the default settings of the engine must be used to determine the settings of the <invoke> activity. That is, partnerLink, portType, operation, inputVariable, and maybe outputVariable are derived by these default settings. The script itself is performed when the appropriate Web service of the process engine in invoked. |
| Embedded Process | This will map to a <scope> element. The scope is not an independent <process> and will share the process variables of the higher-level process. |

| | |
|---|---|
| Sub-Process Activity | BPEL does not have a sub-process element. Thus Independent Sub-Processes MUST map to a BPEL <process>; the contents of the Sub-Process will be contained within a separate process. The Sub-Process object itself maps to an <invoke> activity that “calls” the process.<br><br>BPEL does not support Reference type of Sub-Processes. However, the Sub-Process will be used as a placeholder for the Sub-Process that will be mapped.<br><br>Mapping for an Independent Sub-Process:<br><br>The **DiagramRef** and **ProcessRef** attributes will identify the process that will be used for the mapping to the BPEL process.<br><br>The **OutputPropertyMaps** attribute of the referenced process maps to the inputVariable attribute of the <invoke> activity.<br><br>The **InputPropertyMaps** attribute of the referenced process maps to the outputVariable attribute of the <invoke> activity.<br><br>See the Start event above for more information about how **Simple Interaction** maps to BPEL and WSDL.<br><br>Mapping for a Reference Sub-Process:<br><br>The **SubProcessRef** attribute references another Sub-Process in the Process. It is the referenced Sub-Process that will be mapped and the mappings will be placed in the location of the Reference Sub-Process; another copy of the entire mapping will be created in this location (the mappings will also exist in the referenced Sub-Process’ original location). |
| **Course Control Part** | **Course Control Part** will map to a variety of BPEL elements (e.g., <if>, <pick>, <flow>) and patterns of elements.<br><br>**Course Control Part** potentially marks the end of a BPEL structured element, if the correct number of flows converge.<br><br>The elements that follow **Course Control Part**, until all the outgoing paths have converged, will be contained within the extent of the mapping (e.g., they will be placed within a <sequence> within an <if><condition> and any number of <if><elseif><condition>s). |
| Exclusive Split<br><br>Exclusive Join | **Exclusive Split** will map to <if>.<br><br>Each **Gate** will map to branches specified by <if> and <elseif> (within <if>). The order of branches is maintained.<br><br>Each **guard** association between **Succession** and **Condition** associated with the Gates will map to <condition> elements within <if> or <elseif>.<br><br>The Default Gate (**ExclusiveSplit.default**) will map to the <else> element of <if>.<br><br>If there is more than one element that follows the Gate or the Default Gate, including assignments, then these elements will be placed inside a <sequence>. |

| | |
|---|---|
| **Embedded Process** with an **Change Condition Step connected by a Racing Connection** | This will map to &lt;pick&gt;. The elements of the &lt;pick&gt; will be determined by the targets of the outgoing **Processing Succession**.<br><br>If the **Instantiate** attribute is set to False, the createInstance attribute of the &lt;pick&gt; MUST NOT be included OR it MUST be set to “no.”<br><br>If the **Instantiate** attribute is set to True, the createInstance attribute of the &lt;pick&gt; MUST NOT be included OR it MUST be set to “yes.”<br><br>If the target is a **Simple Activity** with an incoming **Simple Interaction**, it maps to an &lt;onMessage&gt; element within &lt;pick&gt;.<br><br>The attributes of the **Simple Activity** will map to the appropriate elements of the &lt;onMessage&gt;, such as operation and portType.<br><br>If there is a **Time Change Condition** on **Succession**, the activity maps to an &lt;onAlarm&gt; element within &lt;pick&gt;.<br><br>**TimeChange.timeExpression** maps to the until attribute of the &lt;onAlarm&gt;.<br><br>**Cycle Change.**timeExpression maps to the for attribute of the &lt;onAlarm&gt;.<br><br>If there is a **Fact Change Condition** on **Succession**, the event will be considered as the same as receiving a message from a system that tracks and generates Rule events. Thus, this will map to an &lt;onMessage&gt; element within the &lt;pick&gt;.<br><br>If there is more than one element that follows the first target, including assignments, then these elements will be placed inside a &lt;sequence&gt; activity. |
| Parallel Split<br><br>Parallel Join | This will map to &lt;flow&gt;. |

| | |
|---|---|
| Inclusive Split<br>Inclusive Join | **Inclusive Split** will map to a set of <if>s within a <flow>. An additional <if> will be required if the Default Gate (**InclusiveSplit.default**) is used.<br>Each **Gate** will map to <if>, which is binary in nature, i.e. has only the main <if> branch and the <else>.<br>Each **guard** association between **Succession** and **Condition** associated with the Gates will map to <condition> elements within <if> or <elseif>.<br>If the Default Gate is used, the mapping to BPEL is more complicated, as the decision about whether the Default Gate should be taken will occur after all the other gate decisions have been determined. Only if no other path is taken, will the default path taken. This means that the <if> for the Default Gate will follow the <flow> activity generated for all the Gates of the Gateway. Also, a <sequence> activity must encompass the <flow> and the <if>.<br>If the Default Gate is not used, the <else> element for each <if> will contain an <empty> activity.<br>A <variable> must be used so that the <if> for the Default Gate will know whether or not the default path should be taken. To do this, a BPEL <variable> must be created with a derived name and will have a structure as follows:<br><variable  name="[Gateway.Name]_noDefaultRequired"<br>messageType="noDefaultRequired" /><br>The messageType, type or element attribute is used to specify the type of a variable. Exactly one of these attributes MUST be used. The messageType attribute of the variable element refers to a WSDL message type definition. Thus, the messageType will share the same Name and a corresponding WSDL message must be created. Attribute type refers to an XML Schema type (simple or complex). Attribute element refers to an XML Schema element.<br>If a WSDL <message> element is created to support this variable, the message will be structured as follows:<br><message name="noDefaultRequired" ><br><part name="noDefault" type="xsd:boolean" /><br></message><br>An <assign> activity will be created to initialize the <variable> before the start of the loop. This <assign> precedes the <flow> activity that contains all the <if>s derived from the Gates. This will be the first activity within the <sequence> activity. The <assign> will be structured as follows:<br>If any of the <if>s within the <flow> passes the condition of the <if>, then the noDefaultRequired must be set to True. This will ensure that the Default Gate will bypass the mapped activities for the Default Gate.<br>An <assign> activity will be created to set the variable to True. This will be the last activity within the <sequence> activity within the switch. The <assign> will be structured as follows: |

| | |
|---|---|
| | The &lt;condition&gt; for the &lt;if&gt; will use the noDefaultRequired variable and will be structured as follows:<br>`<if>`<br>`<condition>`<br>`bpel:getVariableProperty(`<br>`[Gateway.Name]_noDefaultRequired,`<br>`noDefault)=true`<br>`</condition>`<br>`<sequence>`<br>`<!--The mappings of the original activity are placed here.-->`<br>`<!--An assign activity (see below) is placed here.-->`<br>`</sequence>`<br>`<else>`<br>`<empty/>`<br>`</else>`<br>`</if>`<br>If there is more than one element that follows the first target, including assignments, then these elements will be placed inside a &lt;sequence&gt; activity.<br>`<assign name="[Gateway.Name]_set_noDefault">`<br>`<copy>`<br>`<from>true</from>`<br>`<to   variable="[Gateway.Name]_noDefaultRequired"`<br>`part="noDefault" />`<br>`</copy>`<br>`</assign>` |
| Complex Split<br>Complex Join | N/A |

## 7.7 Flows

| BPDM | BPEL |
|---|---|
| Processing Succession | This MAY map to a &lt;link&gt; element. In many situations, however, **Processing Succession** will not map to a &lt;link&gt; element; to connect activities that are listed in a BPEL structured activity (e.g., a &lt;sequence&gt;), the &lt;link&gt; elements are not required. &lt;link&gt; elements are only appropriate when the **Processing Successions** are Connecting Objects within a &lt;flow&gt;. However, only the **Processing Successions** that are completely contained within the boundaries of the &lt;flow&gt; will be mapped to a &lt;link&gt;. |

| | |
|---|---|
| | If another structured activity (e.g., a <while>) is contained within the flow, then the **Processing Successions** that would be appropriate for the contents of the structured activity, would not be mapped to a <link>.<br>If the **Processing Succession** is being mapped to a <link>:<br>• The Name attribute of the Process (**NamedElement.name**) SHALL map to *name* attribute of the <link>. The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute.<br>• The mapping of the source activity will include a <source> element.<br>• The Name of the **Processing Succession** (**NamedElement.name**) will map to *linkName* attribute of the <source> element. The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *linkName* attribute.<br>• If the source object is a **Course Control Part** and it maps to an activity, the mapping is the same as if the source object is an activity (see above).<br>• If the **Course Control Part** does not map to an activity, the **Processing Succession** will be combined with one of the **Course Control Part**'s incoming **Processing Succession**s. (There will be a separate <link> for each of the incoming **Processing Succession**s). The source of the second **Processing Succession** will be used at the source for the original **Processing Succession**. Then this mapping is the same as if the source object is an activity (see above).<br>• The mapping of the target activity will include a <target> element.<br>• The Name of the **Processing Succession** (**NamedElement.name**) will map to *linkName* attribute of the <target> element. The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *linkName* attribute.<br>• If the target object is a Gateway and it maps to an activity, the mapping is the same as if the target object is an activity (see above).<br>• If the **Control Course Part** does not map to an activity, the **Processing Succession** will be combined with one of the **Course Control Part**'s outgoing **Processing Succession**s. (There will be a separate <link> for each of the outgoing **Processing Succession**s). The target of the second **Processing Succession** will be used at the target for the original **Processing Succession**. Then this mapping is the same as if the target object is an activity (see above). |
| **Processing Succession** with **Condition** | A <flow> will be required and the **Processing Succession** will map to a <link> element. An <empty> activity will be placed in the flow and will contain all the <source> elements. The **Condition** will then map to the transitionCondition attribute of the <source> element that is contained in the appropriate BPEL activity.<br>The mapping of **Processing Succession** with **Condition** when the source object is a **Course Control Part** is described in **Exclusive Split/Join** and **Inclusive Split/Join**. |

| | |
|---|---|
| ExclusiveSplit.default<br>InclusiveSplit.default | See **Exclusive Split/Join** and **Inclusive Split/Join**. |
| **Processing Succession** from the **errorPart Behavioral Change Part** | All the activities that follow the **Processing Succession**, until the Exception Flow merges back into the Normal Flow, will be mapped to BPEL and then placed within the &lt;faultHandlers&gt; element for the &lt;scope&gt; of the activity (and usually within a &lt;sequence&gt;).<br>If there is only one activity in the &lt;faultHandlers&gt; element for the scope of the activity, then this activity will be placed within a &lt;sequence&gt; and preceded by an &lt;assign&gt; (as described below).<br>The mapping of the original activity will be placed within a &lt;sequence&gt; (if it had not been already). The original activity will be followed by an &lt;if&gt;, with one &lt;condition&gt; and an empty &lt;else&gt; as follows:<br>&lt;if&gt;<br>&lt;condition&gt;<br>bpel:getVariableProperty(<br>[activity.Name]_normalCompletion,      normalCompletion)=true<br>&lt;/condition&gt;<br>&lt;sequence&gt;<br>&lt;!--The mappings of the Process activities until the merging of the Exception Flow are placed here.--&gt;<br>&lt;/sequence&gt;<br>&lt;else&gt;<br>&lt;empty/&gt;<br>&lt;/else&gt;<br>&lt;/if&gt; |

| | |
|---|---|
| | A <variable> must be used so that the <if> will know whether or not the Exception Flow or Normal Flow had reached that point in the Process. It must be created with a derived name and will have structure as follows:<br>`<variable name=”[activity.Name]_normalCompletion”`<br>`messageType=”noDefaultRequired” />`<br>The messageType, type or element attribute is used to specify the type of a variable. Exactly one of these attributes MUST be used. The messageType attribute of the variable element refers to a WSDL message type definition. Thus, the messageType will share the same Name and a corresponding WSDL message must be created. Attribute type refers to an XML Schema type (simple or complex). Attribute element refers to an XML Schema element.<br>If a WSDL <message> element is created to support this <variable>, the message will be structured as follows:<br>`<message name=”noDefaultRequired” >`<br>`<part name=”normalCompletion” type=”xsd:boolean” />`<br>`</message>`<br>The <assign> will be created to initialize the <variable> before the start of the original activity. The <assign> will be structured as follows:<br>the <sequence> activity of the <faultHandlers>. The <assign> will be structured as follows:<br>`<assign name=”[activity.Name]_initialize_normalCompletion”>`<br>`<copy>`<br>`<from>true</from>`<br>`<to variable=”[activity.Name]_normalCompletion”`<br>`part=”normalCompletion” />`<br>`</copy>`<br>`</assign>`<br>If a fault is thrown and <faultHandlers> is activated, then an <assign> activity will be used to set the <variable> to False. This will be the first activity within the <sequence> activity of the <faultHandlers>. The <assign> will be structured as follows:<br>`<assign name=”[activity.Name]_set_normalCompletion”>`<br>`<copy>`<br>`<from>false</from>`<br>`<to variable=”[activity.Name]_normalCompletion”`<br>`part=”normalCompletion” />`<br>`</copy>` |
| **Simple Interaction** | No specific mapping to a BPEL element. It represents a message that is sent through a WSDL operation that is referenced in a BPEL <receive>, <reply>, or <invoke>.<br>See Start, Intermediate and End Events for mappings pertaining to **Simple Interaction**. |

| | |
|---|---|
| **Change Condition Step** monitoring **Compensation** | See **Compensation Connection** in Intermediate Events. |

## 7.8 Additional Constructs

| **BPDM** | **BPEL** |
|---|---|
| **Activity** with **Conditional Loop** | This will map to a <forEach> activity. The <forEach> iterates its child <scope> activity exactly N+1 times where N equals the <finalCounterValue> minus the <startCounterValue>. |
| **Activity** with **For Loop** or **Multi Instance Loop** | A Multi Instance Loop can be either sequential or parallel. **MultiInstanceLoop.ordering** maps to the parallel (=”yes\|no”) attribute of <forEach>.<br><br>A sequential MI loop maps to <forEach> as in Basic Loop above so that forEachCount equals to N + 1.<br><br>Four flow conditions (None \| One \| All \| Complex) exist for parallel multi-instance loops:<br><br>None – There is no synchronization or control of the Tokens that are generated through the multi-instance activity. Each Token will continue on independently and each Token will create a separate instantiation of each activity they encounter. Basically, there is a separate copy of the whole process, for each copy of the MI activity, after that point. Each copy of the remainder of the process will continue independently.<br><br>One – Only one of the spawned processes must be completed before the original process can continue.<br><br>All – All of the spawned processes must be completed before the original process can continue.<br><br>Complex – The difference from All is that the number of completed spawned processes required before the process flow will continue must be determined and the processes have been completed. |

| | |
|---|---|
| | The BPDM **Activity Loop** is kind of **Embedded Process** that can execute its content multiple times. Upon completion of each iteration the activity loop will generate **Iteration Finish** happening. This happening can be used in the outgoing **Succession**s to specify that a Succession should be activated on loop iteration completion. Depending on the flow condition:<br>• None – **Succession** on **Iteration Finish** of **Activity Loop**<br>• One – **Succession** on **Iteration Finish** of **Activity Loop** with **Succession.guard** evaluating to the string "first iteration only"<br>• All – **Succession** on **Finish** of **Activity Loop**<br>• Complex – on **Iteration Finish** of **Activity Loop** with **Succession.guard** evaluating to a boolean value. If the value is True then the Succession will be followed<br>A <completionCondition> may be used within the <forEach> to allow the <forEach> activity to complete without executing or finishing all the branches specified.<br>The <forEach> activity without a <completionCondition> completes when all of its child <scope>s have completed. The <completionCondition> element is optionally specified to prevent some of the children from executing (in the serial case), or to force early termination of some of the children (in the parallel case).<br>The <branches> element within <completionCondition> represents an unsigned –integer expression used to define a completion condition of the “at least N out of M” form. The actual value B of the expression is calculated once, at the beginning of the <forEach> activity. It will not change as the result of the <forEach> activity’s execution. At the end of execution of each directly enclosed <scope> activity, the number of completed children is compared to B, the value of the <branches> expression. If at least B children have completed, the <completionCondition> is triggered: No further children will be started, and currently running children will be terminated.<br>The mapping to BPEL per flow condition is as follows:<br>• None – This is not supported by <forEach>. To create this behavior, the remainder of the process will be moved into a new derived <process>. This process will be spawned through a one-way <invoke> that will be placed within the <while> activity.<br>• One – <completionCondition> evaluates to 1.<br>• All – No <completionCondition> specified.<br>• Complex – <completionCondition> evaluates to B (1 < B < N + 1). |

| | |
|---|---|
| Holder | A BPDM Process can define multiple **Holder** objects. A BPDM **Holder** specializes **TypedElement** and thus can define the type of the value it can hold.<br>**Holder** maps to a BPEL <variable>.<br>BPEL uses three kinds of variable declarations: WSDL message type, XML Schema type (simple or complex), and XML Schema element.<br>In the case of WSDL variable declaration, the <variable> element will be structured as follows:<br>&lt;variable<br>name="[Process.Name]_Data"<br>messageType=<br>"[Process.Name]_ProcessDataMessage" /><br>The <message> element will be structured as follows:<br>&lt;message name="[Process.Name]_ProcessDataMessage"><br>&lt;part name="[Property.Name]"<br>type="xsd:[Property.Type]" /><br>&lt;/message> |
| Transaction | Open issue |
| Part Group | A <scope> provides the context which influences the execution behavior of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler. Contexts provided by <scope> activities can be nested hierarchically, while the root context is provided by the <process> construct. |
| **Comment** from UML2 infrastructure | Can map to the <documentation> element. If the **Comment** is associated with an object that has a straight-forward mapping to a BPEL element, then the text of the **Comment** will be placed in the <documentation> element of that object. If there is no straight-forward mapping to any element, then the text will be appended to the <documentation> element of the <process>. |
| Simple Interaction.transformation | This will map to BPEL <assign> activities. |

# 8 Proof of Concept Language Mappings

The following sub-sections describe mappings to specific languages as proofs of concept.

## 8.1 BPEL Mapping

[To be completed in a later version of this specification.]

## 8.2 WS-CDL Mapping

[To be completed in a later version of this specification.]