

A Reflective Metamodel of Object-Process Methodology: The System Modeling Building Blocks

Iris Reinhartz-Berger

University of Haifa

Carmel Mountain, Haifa 31905, Israel

Phone number: 972-4-8288502

Fax number: 972-4-8288522

Email: iris@mis.hevra.haifa.ac.il

Dov Dori

Technion, Israel Institute of Technology

Technion City, Haifa 32000, Israel

Phone number: 972-4-8294409

Fax number: 972-4-8295688

Email: dori@ie.technion.ac.il

A Reflective Metamodel of Object-Process Methodology: The System Modeling Building Blocks

ABSTRACT

In this chapter, we introduce a highly expressive, self-contained reflective metamodel of Object-Process Methodology (OPM). OPM enables universal system modeling based on the notions of processes that transform objects. Extending the object-oriented approach, which views processes as residents of objects, OPM provides for the existence of stand-alone processes that can represent transformations in complex systems such as businesses, aircrafts, or organisms. A system modeling and development methodology, which is a combination of a language for expressing the universal (or domain) ontology and an approach for developing systems that uses this language, can be expressed in OPM using objects, processes, and links among them. Through the reflective OPM metamodel, we demonstrate the expressive power of OPM and its applicability as a universal tool for architecting systems that involve structure and dynamics in a highly, intertwined manner.

Keywords: Software Development Methodologies, IS Development Methodologies, Object-Oriented Design, Functional Design, Ontologies, Metamodel, Metamodeling

INTRODUCTION

A system modeling and development methodology is a combination of a *language* for expressing the universal or domain ontology and an *approach* or a *protocol* for developing systems that makes effective use of this language. Metamodeling, the process of modeling a methodology, enables building, understanding, comparing, and evaluating methodologies. The metamodeling process produces a metamodel, i.e., a model of the methodology (metamodel site, 2003). We refer to a methodology that can model itself as a *reflective methodology*, and to metamodeling of a reflective methodology as *reflective metamodeling*.

In other words, a reflective metamodel is defined exclusively in terms of the modeled methodology. A reflective methodology is especially powerful since it is self-contained, so it does not require auxiliary means or external tools to model itself. Object-Process Methodology (OPM), which is a holistic system modeling, development and evolution approach that combines object-oriented notations with process-oriented concepts, is a reflective methodology.

As noted, metamodels have become important means for comparing and evaluating methodologies and their supporting CASE tools. By and large, metamodels are structure- or object-oriented, and hence pertain only to the static elements and relations of the methodology. They therefore do not include the procedural parts of the methodology (also known as "the software process"). Rather, these are usually described loosely and informally in some natural language, most often English. The main reason for this omission of the methodology's "process" part is the lack of expressive power of the methodology to seamlessly and straightforwardly describe not only objects and structure but also processes and behavior.

Object-Process Methodology (OPM) overcomes this shortcoming by treating objects and processes as two equally important entities rather than viewing object classes necessarily as superiors to and owners of processes. Through the bimodal OPM model presentation of Object-Process Diagrams (OPDs) and Object-Process Language (OPL) sentences, this chapter presents the reflective metamodel of the language and notation parts of OPM, namely its semantics and syntax. The other part of the reflective OPM metamodel, which specifies OPM-based system development and evolution processes, can be found in (Dori 2002, pp. 289-309; Dori and Reinhartz-Berger, 2003). A major significance of this work is that it lays out a comprehensive, generic, and formal definition of OPM that enables domain-independent modeling of complex systems, in which structure and behavior are intertwined

and hard to separate. Indeed, real-life systems of interest can almost always be characterized as such.

The chapter is structured as follows. First, the main metamodeling concepts are defined and existing metamodeling approaches are reviewed. Then, the main concepts of OPM are introduced and exemplified through a business enterprise model that handles customer orders and retailer requests. The main part of the chapter is the OPM reflective metamodel, including all its elements, entities, and structural, procedural, and event links. Finally, the contribution of OPM as a universal business modeling methodology is summarized, emphasizing its role in defining new methodologies.

REFLECTIVE METHODOLOGIES AND REFLECTIVE METAMODELING

System analysis and design activities can be divided into three types with increasing abstraction levels: real world, model, and metamodel (Van Gigch, 1991). The real world is what system analysts perceive as reality or what system architects wish to create as reality. A model is an abstraction of this perceived or contemplated reality that enables its expression using some approach, language, or methodology. A metamodel is a model of a model, or, more accurately, a model of the modeling methodology (metamodel site, 2003). Metamodels help understand the deep semantics of a methodology as well as relationships among concepts in different languages or methods. They can therefore serve as devices for methods development, also referred to as methods engineering (Nuseibeh et al., 1996; Rossi, et al., 2000), and as conceptual schemas for repositories of software engineering and CASE tools.

Metamodeling is the process that creates metamodels. The level of abstraction at which metamodeling is carried out is higher than the level at which modeling is normally done for the purpose of generating a model of a system (Henderson-Sellers and Bulthuis, 1998).

The proliferation of object-oriented methods has given rise to a special type of metamodeling—reflective metamodeling, i.e., modeling a methodology using its own means

alone. While metamodeling is a formal definition of the methodology, reflective metamodeling can serve as a common way to check and demonstrate the methodology's expressive power.

Existing object-oriented languages, notably the standard Unified Modeling Language (UML), have partial reflective metamodels. The reflective UML metamodel in (Object Management Group, 2001), for example, includes class diagrams; OCL (Object Constraint Language) (Warmer and Kleppe, 1999) constraints, which are added on top of the UML graphics as a textual means to express constraints; and natural language explanations for describing the main elements in UML and the static relations among them. This metamodel is incomplete in more than one way. First, UML is only a notation and not a methodology, so only the language elements are metamodeled, but not any (object-oriented or other) development process. Second, class diagrams are used to model all ten UML views (diagram types) and the metamodel does not enforce complete consistency requirements among the various views of a UML system model. Third, most of the metamodel (structural) constraints are expressed in OCL, which is a programming-language-like add-on to UML.

The Meta Object Facility (MOF) (Object Management Group, 2003) is a standard metadata architecture whose main theme is extensibility and support of metadata. MOF defines four layers of metadata: information (i.e., real world concepts, labeled M0), model (M1), metamodel (M2), and meta-metamodel (M3). The meta-metamodel layer describes the structure and semantics of meta-metadata. In other words, it is an "abstract language" for defining different kinds of metadata (e.g., meta-classes and meta-attributes).

The Meta Modeling Facility (MMF) (Clark et al., 2002) provides a modular and extensible method for defining and using modeling languages. It comprises a static, object-oriented language (MML) to write language definitions, a tool (MMT) to interpret those definitions,

and a method (MMM), which provides guidelines and patterns encoded as packages that can be specialized to particular language definitions.

MOF and MMF have been applied to metamodel UML. Since both are object-oriented, they emphasize UML elements, while the procedural aspects are suppressed. Since OPM combines the object- and process-oriented approaches in a single framework, it can specify system structure and dynamics in a balanced way. In particular, metamodels expressed in OPM capture both the language and the system development approach parts of the modeled methodology.

OBJECT-PROCESS METHODOLOGY IN A NUTSHELL

Object-Process Methodology (OPM) (Dori, 2002) is a holistic approach to the modeling, study, development, and evolution of systems. Structure and behavior coexist in the same OPM model to enhance the comprehension of the system as a whole. Contrary to UML with its ten diagram types, OPM shows the system's structure and behavior in the same and single diagram type, enabling direct expression of relations, interactions, and effects. This trait reinforces the users' ability to construct, grasp, and comprehend the system as a whole and at any level of detail. Moreover, Soffer et al. (2001) concluded that OPM is ontologically complete according to the Bunge-Wand-Weber (BWW) evaluation framework (Wand and Weber, 1993). The BWW framework aims to be a theoretical foundation for understanding the modeling of information systems. Any modeling language (or grammar) must be able to represent all things in the real world that might be of interest to users of information systems, otherwise, the resultant model is incomplete (Rosemann and Green, 2002). Hence, OPM completeness according to the BWW framework is indicative of OPM's expressive power. Appendix A lists the ontological constructs of information systems, their BWW explanations, and their OPM representation as indicted in (Soffer et al. 2001).

Due to its structure-behavior integration, OPM provides a solid basis for modeling complex systems. Indeed, OPM has been extended to support the modeling of common types of systems, including real-time systems (Peleg and Dori, 1999), ERP (Soffer et al., 2003), and Web applications (Reinhartz-Berger et al., 2002). Three independent experiments showed that OPM is more comprehensible than object-oriented techniques in modeling the dynamic and reactive aspects of real time systems (Peleg and Dori, 2000), Web applications (Reinhartz-Berger and Dori, 2004), and discrete event simulation systems.

OPM Concepts

The elements of OPM ontology are entities and links. Entities generalize things and states. A *thing* is a generalization of an *object* and a *process* – the two basic building blocks of any OPM-based system model. At any point in time, each object is at some *state*, and object states are changed through the occurrence of processes. Analogously, links can also be structural or procedural. *Structural links* express static, structural relations between pairs of objects or processes. These relations hold for the system regardless of the time dimension. Aggregation, generalization, characterization, and instantiation are the four fundamental structural relations. In addition, general structural relations can take on any semantics, which is expressed textually by their user-defined tags.

The behavior of a system is manifested in three major ways: (1) processes can transform (generate, consume, or change) things, (2) things can enable processes without being transformed by them, and (3) things can trigger events that (at least potentially, if some conditions are met) invoke processes. Accordingly, a procedural link can be a transformation link, an enabling link, or an event link.

The complexity of an OPM model is controlled through three scaling (refinement/abstraction) processes: *in-zooming/out-zooming*, in which the entity being refined is shown enclosing its constituent elements; *unfolding/folding*, in which the entity being refined is shown as the root

of a directed graph; and *state expressing/suppressing*, which allows for showing or hiding the possible states of an object. These mechanisms enable OPM to recursively specify and refine the system under development to any desired level of detail without losing legibility and comprehension of the complete system. Each time a diagram is about to get too cluttered, a new diagram can be spawned. The new diagram is linked to and elaborates upon the ancestor diagram.

The Bimodal Graphic-Text Representation of OPM

Two semantically equivalent modalities, one graphic and the other textual, jointly express the same OPM model. A set of inter-related Object-Process Diagrams (OPDs), constitute the graphical, visual OPM formalism. Each OPM element is denoted in an OPD by a dedicated symbol, and the OPD syntax specifies correct and consistent ways by which entities can be connected via structural and procedural links. The Object-Process Language (OPL), precisely defined by a grammar, is the textual counterpart modality of the graphical OPD set. OPL is a dual-purpose language, oriented towards humans as well as machines. Catering to human needs, OPL is designed as a constrained subset of English, which serves domain experts and system architects. All the stakeholders can use the OPL specification along with the corresponding OPDs to jointly engage in analyzing and designing a system. Every OPD construct is expressed by a semantically equivalent OPL sentence or phrase. Designed also for machine interpretation through a well-defined set of production rules, OPL provides a solid basis for automating the generation of the designed application. According to Mayer's cognitive theory (2001), this dual representation of OPM increases the processing capability of humans. Moreover, OPDs constitute a complete and consistent visual formalism that goes hand in hand with the OPL in the following meaning: *Anything that is expressed graphically by an OPD is also expressed textually in the corresponding OPL paragraph, and vice versa.*

OPCAT (Dori et al., 2003), a Java-based Object-Process CASE Tool, automatically translates each OPD into its equivalent OPL paragraph (collection of OPL sentences) and vice versa.

OPM CONCEPTS DEMONSTRATED BY AN INVENTORY SYSTEM MODEL

Before presenting the OPM reflective metamodel, in this section we explain and demonstrate OPM concepts through an OPM model of a simple business enterprise inventory system which handles orders. This enterprise can get requests for products from individual customers or from retailers. The OPM model of this enterprise, which includes information modeling as well as business process specification, is presented in Figures 1-7 using both OPDs and their corresponding OPL paragraphs. This dual representation increases the model clarity and accessibility, as readers who are familiar with OPM and its graphical notation can use the OPDs, while readers who are new with OPM will probably prefer to start with the OPL paragraphs. Since the graphical and textual notations of OPM are equivalent, and, from a cognitive viewpoint, complementary, the reader can choose the modality (text or graphics) with which he/she is most comfortable and switch between the two at will. Furthermore, the OPL paragraphs are self-documented and hence need no further explanations.

OPM Elements

As noted, OPM consists of two types of elements: entities and links. *Entities* are classified into things and states. A *thing* is a generalization of an object and a process. *Objects* are entities that exist, while *processes* are entities that transform things by generating, consuming, or affecting them. A *state* is a situation at which an object exists. Therefore, a state is not a stand-alone entity, but rather an entity that is "owned" by an object. At any given point in time, the state-owning object is at one of its states. The status of an object, i.e., the current state of the object, is changed solely through an occurrence of a process. Objects and processes are respectively denoted in an OPD by rectangles (as in class diagrams in UML

and earlier notations) and ellipses (as in data-flow diagrams). Following Statecharts (Harel, 1987) notation, the OPD symbol of a state is a rounded corner rectangle within the rectangle of its owning object. In Figure 1, for example, **Order**, **Receipt**, **Product Catalog**, **Customer**, and **Retailer** are objects, while **Ordering** is a process. In Figure 2, **created**, **paid**, **supplied**, and **completed** are states of the **Order Status** attribute.

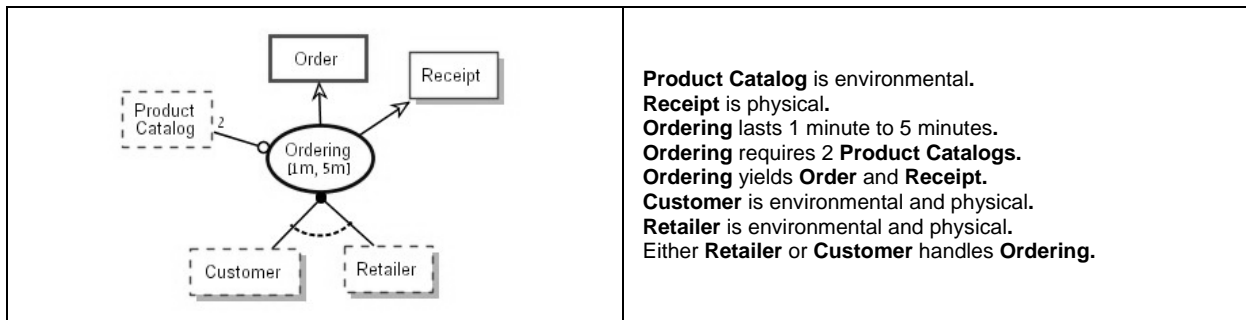


Figure 1. Top level, System Diagram (SD) of the ordering system

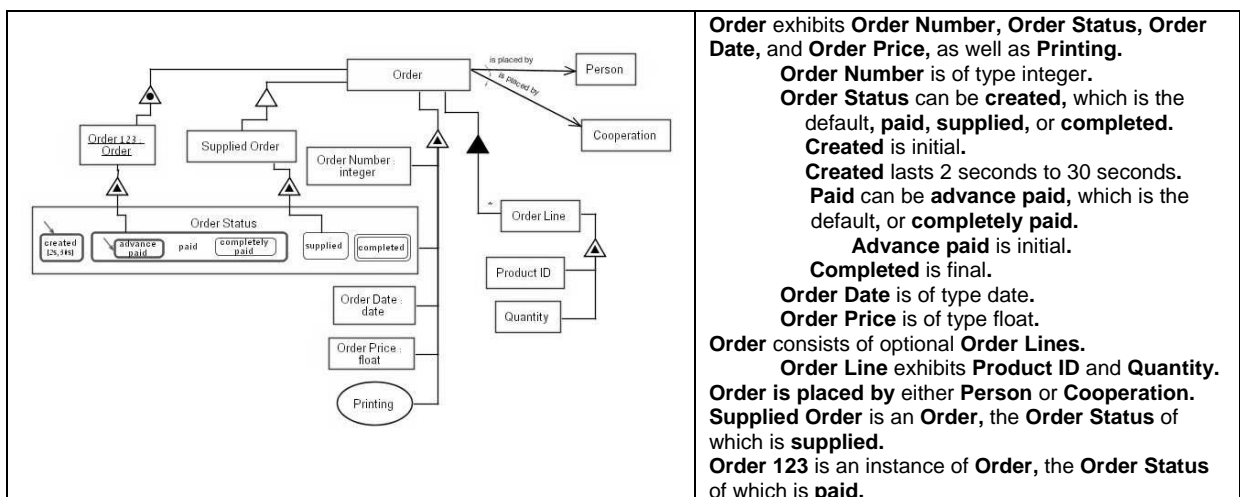


Figure 2. SD1, in which **Order** is structurally unfolded

A *link* is an element that connects two entities to represent some semantic relation between them. Links can be structural or procedural. A *structural link* is a binary relation between two entities, which specifies a structural aspect of the modeled system, such as an aggregation-participation (whole-part) or a generalization-specialization relation.

A *procedural link* connects an entity with a process to denote a dynamic, behavioral flow of information, material, energy, or control. An *event link* is a specialization of a procedural link

which models a significant happening in the system that takes place during a particular moment and might trigger a process if preconditions are met.

Links are denoted in an OPD by lines with different types of arrowheads or triangles, as summarized in Appendix B. In Figure 1, for example, **Ordering**, which is triggered (activated) by either **Customer** or **Retailer**, uses **Product Catalog** as an input, and creates **Order** and **Receipt** as outputs.

Any OPM element can be either systemic or environmental. A *systemic element* is internal to the system and has to be completely specified, while an *environmental element* is external to the system model and may therefore be specified only partially. The OPD symbol of an environmental element differs from its systemic counterpart in that its borderline is dashed. The **Product Catalog** in Figure 1, for example, is an environmental object; it is external to the system but should be used as an unchangeable input for the **Ordering** process.

In an orthogonal fashion, an OPM element can also be either physical or informatical. A *physical element* is tangible in the broad sense, while an *informatical element* relates to information. A physical entity is symbolized in an OPD as a shadowed closed shape – rectangle, ellipse, or rounded corner rectangle for a physical object, a physical process, or a physical state, respectively. The **Receipt** in Figure 1, resulting from the **Ordering** process, is a systemic and physical object, while the **Customer** and the **Retailer** are environmental and physical objects.

OPM Things

As noted, a thing is a generalization of an object and a process. A thing can be simple or complex. A thing is simple if it has no parts, features (attributes or operations), or specializations, and is complex otherwise. An *object* is a thing that exists, at least potentially, and represents a class of instances that have the same structure and can exhibit the same behavior. The **Order** in Figure 2, for example, is a complex object which exhibits four simple

attributes (each of which is an object in its own right): **Order Number**, which is of type integer, **Order Status**, which is of an enumeration type, **Order Date**, which is of type date, and **Order Price**, which is of type float.

A *process* is a class of occurrences (or instances) of a behavior pattern, which transforms at least one thing. Transformation can be creation, consumption, or effect (state change) of a thing (usually an object). To carry out the transformation, the process may need to be enabled by one or more things of different types of classes, which are considered instruments (enablers) for that process. An instrument is a non-human object which is not transformed by the process it enables.

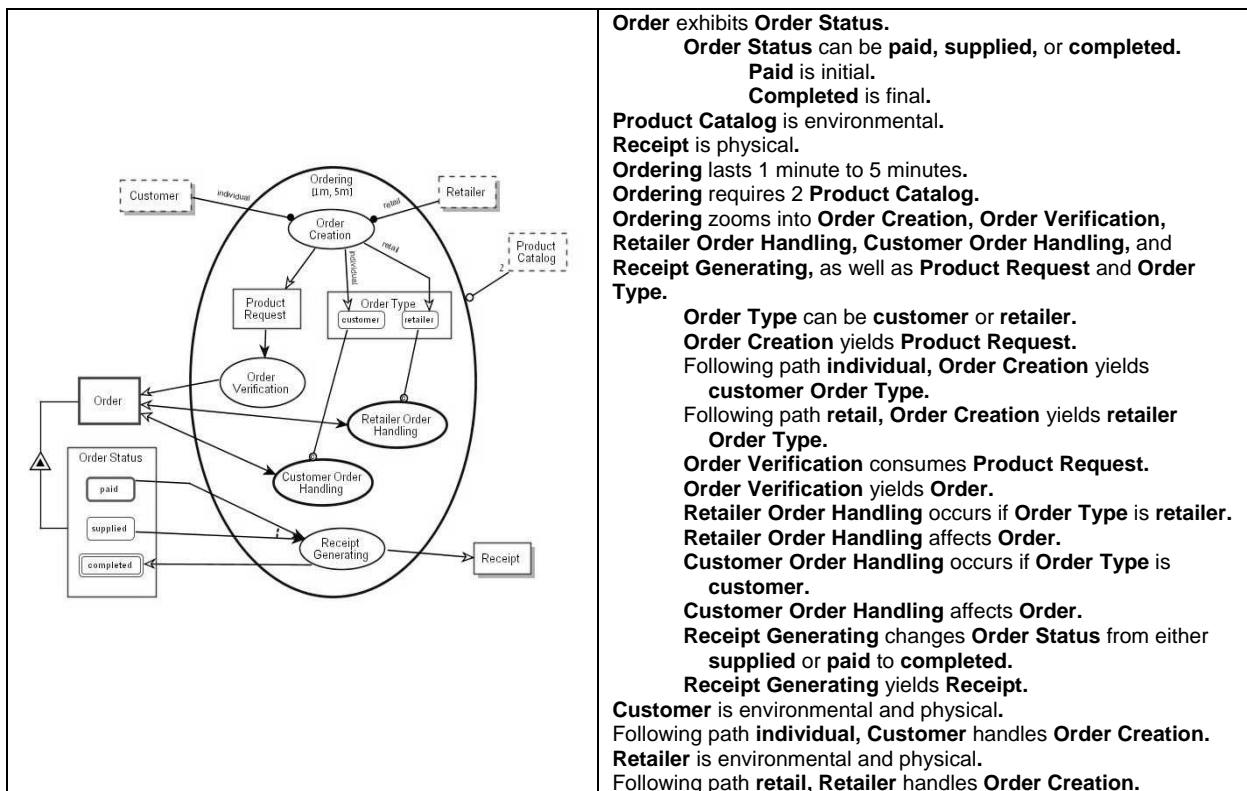


Figure 3. SD2, in which **Ordering** is in-zoomed

Analogous to an object instance, a *process instance* is an occurrence (one-time execution) of the specific process. The execution time of a process can be constrained by minimal and maximal limits, implying that any process execution can only take a time interval that falls within these time limits. The time limits appear in the OPD as [minimal time constraint,

maximal time constraint] within the ellipse representing the process. For example, the specification of the minimal and maximal time limits of the **Ordering** process in Figure 1 and Figure 3 implies that it must take at least 1 minute and at most 5 minutes. The corresponding OPL sentence is “**Ordering** lasts 1 minute to 5 minutes.”

Following the UML notation of classes and objects, a thing instance is denoted in OPM by a rectangle or an ellipse within which the class name is written as “**:ClassName**”. The identifier of the instance can optionally precede the colon. The OPL syntax for an instance makes use of the reserved word “the” in an instance phrase, which is “The **ClassName InstanceName**”. For example, suppose in Figure 3 we replace **Retailer** by **Storex**, an instance of **Retailer**. In the object instance box in the OPD we would write “**Storex: Retailer**”, and instead of the OPL sentence “Following path **retail**, **Retailer** handles **Order Creation**.” we would write “Following path **retail**, the **Retailer Storex** handles **Order Creation**.” If the instance identifier is not explicitly specified, the OPL instance phrase would be “The **ClassName** instance.” In our example the sentence would be “The **Retailer** instance handles **Order Creation**.”

A process can be atomic, sequential, or parallel. An atomic process is a lowest-level, elementary action which is not divided into sub-processes, while sequential and parallel processes are refined (usually through in-zooming) into several sequential or parallel sub-processes. The time line in an OPD flows from the top of the diagram downwards. Hence, the vertical axis within an in-zoomed process defines the execution order: The sub-processes of a sequential process are depicted in the in-zoomed frame of the process stacked on top of each other with the earlier process on top of a later one.

Analogously, sub-processes of a parallel process appear in the OPD side by side, at the same height. In Figure 4 and Figure 5, **Retailer Order Handling** and **Customer Order Handling** are respectively in-zoomed, to show their two sub-processes, **Paying** and **Supplying**. In the in-zoomed version of **Customer Order Handling** (Figure 5), **Paying** and **Supplying** are executed

in a serial order: First, the **Customer** pays and only afterwards **Order** is supplied. In the in-zoomed version of **Retailer Order Handling** (Figure 4), on the other hand, **Paying** and **Supplying** are executed independently and may occur in parallel.

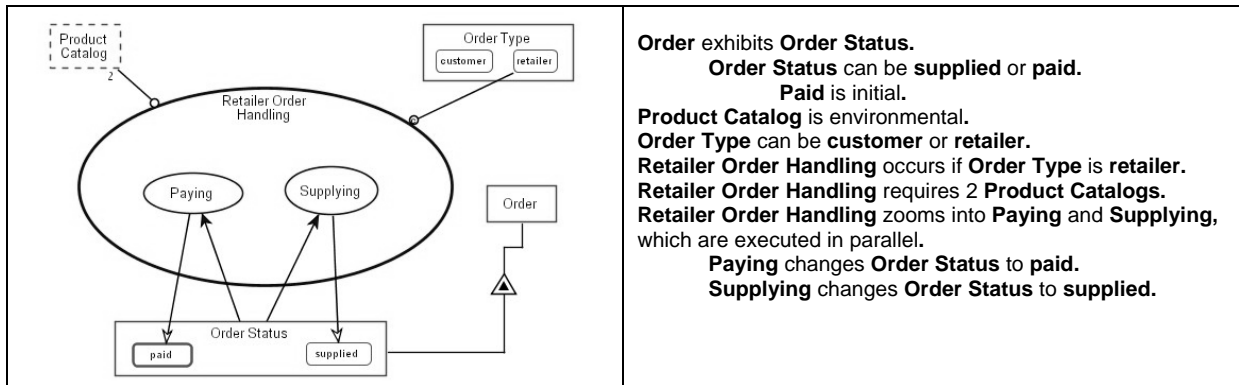


Figure 4. **SD2.1**, in which **Retailer Order Handling** is in-zoomed

The default execution order is the sequential one, so only the parallel execution order is specified in OPL using the reserved phrase “which are executed in parallel”. For example, the in-zooming sentence in Figure 4 is “**Retailer Order Handling zooms into Paying and Supplying, which are executed in parallel.**”

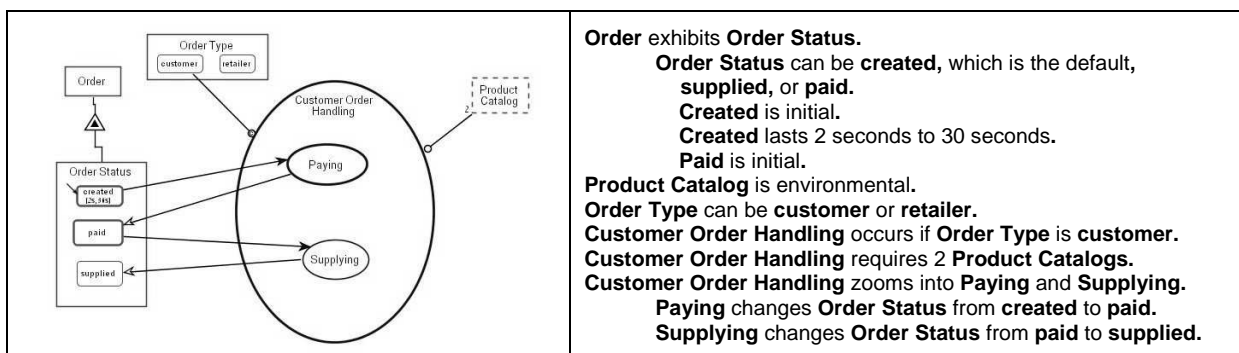


Figure 5. **SD2.2**, in which **Customer Order Handling** is in-zoomed

OPM States

A *state* is a situation in which an object can be for some period of time. At any point in time an object is in exactly one of its states. A state can be a value from a continuous or discrete value range, or a finite enumerated set of named states. **Order Status** in Figure 2, for example, has four possible, top-level states: **created**, **paid**, **supplied**, and **completed**.

A state can be initial, final, or default. Both **created** and **paid** are initial states, as denoted by the thick borderline rounded corner rectangle. This implies that **Order Status** can be generated in either its **created** or **paid** states, but not at both, since at any point in time an object is in exactly one of its states. If not otherwise specified, **Order** will be generated in its **created** state as denoted by the default mark (the small downward diagonal arrow that points towards the **created** state). The **completed** state is the final state of **Order Status**, as denoted in Figure 2 by the double line rounded corner rectangle. When entering this final state, **Order** can be consumed (i.e., destroyed or deleted). The reserved OPL phrases that describe initial, final, and default states are "is initial", "is final", and "which is the default", respectively (see Figure 2). Like process durations, state durations can also be limited on one or both sides. For example, the **created** state of **Order Status** in Figure 2 has a minimal time limit of 2 seconds and a maximal time limit of 30 seconds, implying that between 2 to 30 seconds must pass from the moment **Order Status** enters its **created** state until it exits this state.

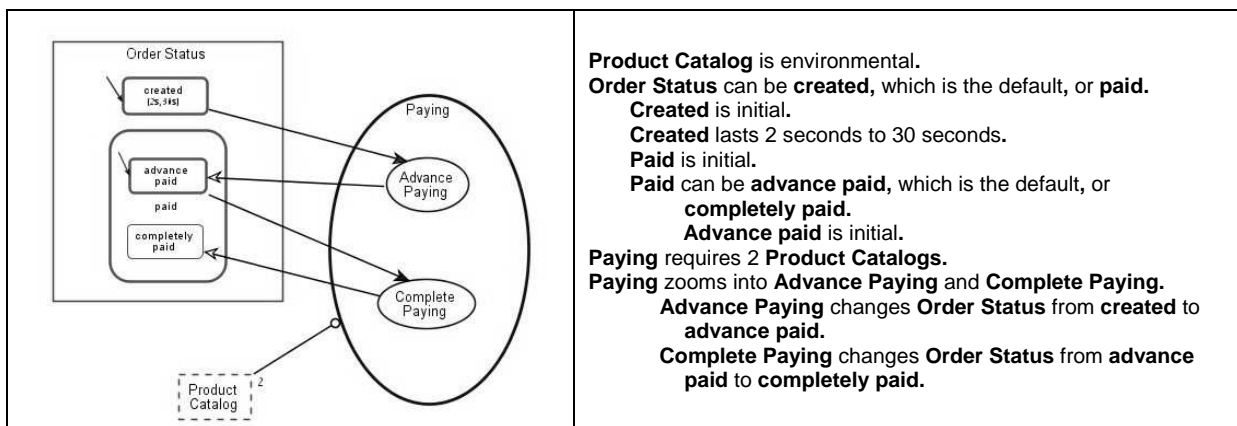


Figure 6. SD2.2.1, in which **Paying** of **Customer Order Handling** is in-zoomed

Like objects and processes, states can be simple or complex. Complex states recursively contain nested states, and the inner composition of a complex state can be exposed by zooming into it. In Figure 2, for example, in its **paid** state, **Order Status** can be at one of two sub-states: **advance paid**, which is the default of a **paid Order**, or **completely paid**. The in-zoomed diagram of **Paying** (of **Customer Order Handling**) in Figure 6 shows that **Advance**

Paying first changes **Order Status** from **created** to **advance paid**, and then **Balance Paying** changes **Order Status** from **advance paid** to **completely paid**.

OPM Links

Links are the "glue" that holds entities (processes and objects with their states) together and enables the construction of system modules of ever growing complexity. OPM links are classified into two types: structural links and procedural links, with the latter specializing into enabling, transformation, and event links.

OPM Structural Links

A *structural link* denotes a structural, i.e., a static, time-independent relation between two elements. It usually connects two objects, but it can also connect two processes. Structural links further specialize into general (tagged) structural links, and four fundamental structural links. A *tagged structural link* can be unidirectional, graphically symbolized by \rightarrow , or bidirectional, graphically symbolized by \leftrightarrow . It is usually labeled by a textual forward tag (for the unidirectional link) or a pair of forward and backward tags (for the bidirectional link). These tags are set by the system architect to convey a meaningful relation between the two linked entities. In Figure 2, for example, the two objects **Order** and **Person** are linked with a general unidirectional, structural link tagged "**is placed by**", connecting an **Order** with the **Person** who placed it. Similarly, **Order** and **Cooperation** are linked with a tagged unidirectional, structural link that is also labeled "**is placed by**".

The four most prevalent and useful OPM structural relations are termed *fundamental structural relations* and are assigned various triangular symbols placed along the line linking the two things. These symbols are graphically more distinct and appealing to the eye than their text tag counterparts. The fundamental structural links are:

1. **Aggregation-Participation** denotes the fact that a thing aggregates (i.e., consists of, or comprises) one or more (lower-level) things, each of which is a part of the whole. It is denoted by \blacktriangle , an equilateral triangle whose tip is linked to the whole and whose base is linked to the parts. To achieve the same semantics, we could use "**consists of**" and "**is part of**" as the forward and backward tags of a tagged bi-directional, structural link, respectively, but, as noted, using the black triangle symbol helps distinguish this relation from any other tagged structural relation (and the other three fundamental structural relations). In Figure 2, **Order** consists of optional (0 or more) **Order Lines**, as the multiplicity constraint * denotes.
2. **Exhibition-Characterization** denotes the fact that a link or a thing exhibits, or is characterized by, another lower-level thing. The exhibition-characterization symbol is \blacktriangleleft . The exhibitor is linked to the tip of the triangle, while the features (which can be attributes or operations) are connected to its base. In Figure 2, **Order** exhibits (i.e., is characterized by) the attributes **Order Number**, **Order Status**, **Order Date**, and **Order Price** and the operation **Printing**, while **Order Line** exhibits **Product** and **Quantity**.
3. **Generalization-Specialization (Gen-Spec)** is a fundamental structural relation between two entities, denoting the fact that the specialized entities share common features, states, and structural and procedural links with the generalizing entity. The symbol of the gen-spec relation is \triangle , a blank triangle whose tip is linked to the generalizing entity and its base – to the specialized entities. In Figure 2, **Supplied Order** defines a sub-class of **Orders** whose status is **supplied**. Like **Order**, **Supplied Order** has its **Order Number**, **Order Status** (which is always supplied), **Order Date**, **Order Price**, **Order Lines**, and an owning **Person** or **Cooperation**. It can also execute the operation **Printing**.
4. **Classification-Instantiation** represents a fundamental structural relation between a class of things and an instance of that class. This type of link is denoted by \blacktriangleleft , a triangle

enclosing a solid circle, the tip of which is linked to the class, while its base – to the instances. **Order 123** in Figure 2 is an instance of an **Order** whose status is **paid**.

Structural links of the same type can be connected by “OR” and “XOR” logical relations to specify alternative structures. An “OR” relation is symbolized by a double dashed arc connecting the relevant structural links, while a “XOR” relation is denoted by a single line, dashed arc. In Figure 2, for example, an **Order is placed by** either a **Person** or **Cooperation**, but not by both. If there were no arcs in that specification, a specific **Order** would have an owning **Person** *and* an owning **Cooperation**.

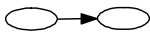
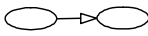
OPM Procedural Links

A procedural link represents a dynamic relation between a process and an entity. Procedural links are divided into enabling links, transformation links, and event links. An *instrument link* is an enabling link that connects a process with an enabler of that process. The enabler is an entity that must be present in order for that process to occur, but it is not transformed as a result of the process occurrence. The instrument link can originate from an object, a process, or a state, denoting that the object existence, the process existence, or the object in the specific state is the enabler, respectively. Graphically, an instrument link is symbolized by $-\circ$, while textually it is represented by the reserved word “requires”. In Figure 1, for example, **Product Catalog** is required for the **Ordering** process. However, the occurrence of **Ordering** does not affect **Product Catalog** in any way. Therefore, **Product Catalog** is an instrument of the process **Ordering**. It is, however, possible that for another process, such as **Catalog Updating**, **Product Catalog** would be an affectee, i.e., an object affected by **Catalog Updating**. Hence, being an instrument for a certain process class can be thought of as a “role” of a thing class with respect to that particular process class.

A *transformation link* denotes that a thing is transformed by the occurrence of a process. Transformation is a generalization of consumption, result, and effect. A *consumption link* is a

transformation link that connects an entity to a process which consumes it. A consumption link is denoted by \rightarrow from the consumed entity to the process, while the reserved word “consumes” represents it in OPL. In Figure 3, for example, **Product Request** is an object that is internal to **Ordering** (in object-oriented programming terms it can be thought of as a local variable of the method **Ordering**) and hence it appears in the in-zoomed frame of **Ordering**. **Product Request** is consumed by the process **Order Verification**. In other words, **Product Request**, which had existed before an occurrence of **Order Verification**, was consumed (destroyed or destructed) by the execution of that process, and it no longer exists after **Order Verification** is over. A consumption link originating from a state of an object means that the process consumes that object only when the object is in that specific state. The corresponding state-specified consumption OPL sentence is “**Process consumes state Object.**”

A *result link* is a transformation link that denotes a creation of a process, an object, or an object at a specific state. It is symbolized in an OPD by \rightarrow from the process to the resultant entity, while the reserved word “yields” denotes it in OPL. In Figure 3, for example, **Order Verification**, which consumed **Product Request**, creates an **Order**. The **Order** had not existed before the beginning of **Order Verification**. Rather, it was created during this execution, and it exists as soon as **Order Verification** is finished.

Since a process is a pattern of behavior or execution, it is also possible for a process to generate or consume not just an object but also a process (e.g., when a process generates a computer program that represents a process). To avoid confusion, the arrowhead pointing at the consuming process is \rightarrow , namely solid (black) rather than blank. Hence,  means that the right process consumes the left one, while  means that the right process yields the left one.

An *effect link* connects a process with a thing that is affected, i.e., undergoes a change, during that process. The effect link, denoted in an OPD by \leftrightarrow where the black arrowhead points

towards the process and the blank arrowhead points towards the affectee (the affected thing), means that the affectee of the process had existed before the process occurred and it continues to exist after the process was finished, but at least one of its states or features has changed.

OPL uses the reserved word “affects” to represent effect links. In Figure 3, for example, **Retailer Order Handling** and **Customer Order Handling** affect **Order**. Figure 4 refines this effect (state change) by explicitly showing that **Paying** of **Retailer Order Handling** changes **Order Status** from any state to **paid** and **Supplying** changes **Order Status** from any state to **supplied**. Figure 5 specifies that **Paying** of **Customer Order Handling** changes **Order Status** from **created** to **paid**, while **Supplying** of **Customer Order Handling** changes **Order Status** from **paid** to **supplied**. These refinements are made possible due to the ability to split an effect link into an input (state consumption) link and an output (state result) link. Overall, the meaning of input and output links can be thought of as “the process consumes the input state and yields the output state”. However, the object as a whole is neither consumed nor generated – it merely changes its state (or its value). Suppressing the object’s states is an abstraction that hides the states, while also joining the input and output links to an effect link.

Procedural links can have multiplicity constraints like their structural counterparts. For example, in Figure 1, **Ordering** requires 2 **Product Catalogs** while affecting one **Order** (the default, when no multiplicity constraint is indicated) and yielding one **Receipt**. Like structural links, procedural links of the same type can be grouped by “OR” and “XOR” connectors to denote different possible instruments, consumees, resultees, and/or affectees of the same process. In Figure 3, for example, **Receipt Generating** can change **Order Status** from either **paid** or **supplied** to **completed**.

A procedural link may have one or more path labels. A *path label* is a character string label on a procedural link that removes the ambiguity arising from multiple procedural links

outgoing from the same entity. When procedural links that originate from an entity are labeled, the one that must be followed is the one whose label is identical with the label of the procedural link through which that entity was reached. The path labels in Figure 3, for example, specify two possible scenarios of **Order Creation**. Symbolized by the path label **individual**, this process occurs at the **Customer** request and it creates a temporary **Order Type** object at state **customer**. Symbolized by the path label **retail**, the process occurs at the **Retailer** request and it creates the temporary **Order Type** object in its **retailer** state. The **Product Request** is generated in both scenarios. The **Customer Order Handling** and **Retailer Order Handling** processes occur according to the **Order Type**, as the conditional enabling links (the instrument links with the letter ‘c’ inside them) denote. A conditional enabling link specifies a branching control construct. If these links were replaced by regular enabling (i.e., instrument) links, the semantics would be “wait until **Order Type** is in its **retailer** state and then execute **Retailer Order Handling**. Afterwards, wait until **Order Type** is in its **customer** state and then execute **Customer Order Handling**.”

Any type of procedural link (except for the result link) can be made conditional. Graphically, this is done by adding the letter ‘c’ to the link symbol, as shown in Appendix B. In OPL, a conditional procedural link is specified by two sentences: one for its procedural aspect (e.g., an enabling sentence: “**Process** requires **Object**.”) and the other is a condition sentence. The two possible condition sentences are a thing condition sentence: “**Process** occurs if **Thing** exists.” and a state condition sentence: “**Process** occurs if **Object** is **state**.”

OPM Event Links

An event is a significant happening in the system that takes place during a particular moment in the system’s lifecycle, and it often triggers some process in the system. An event is represented by an *event link*, which is a procedural link that connects a source entity with a destination process. Following the Event-Condition-Action paradigm, the semantics of an

event link is that the source entity attempts to trigger the destination process. The process does not start unless the event link is enabled, i.e., the event occurs, and all the process' pre-conditions, represented by the regular (conditional or non-conditional) procedural links, are satisfied.

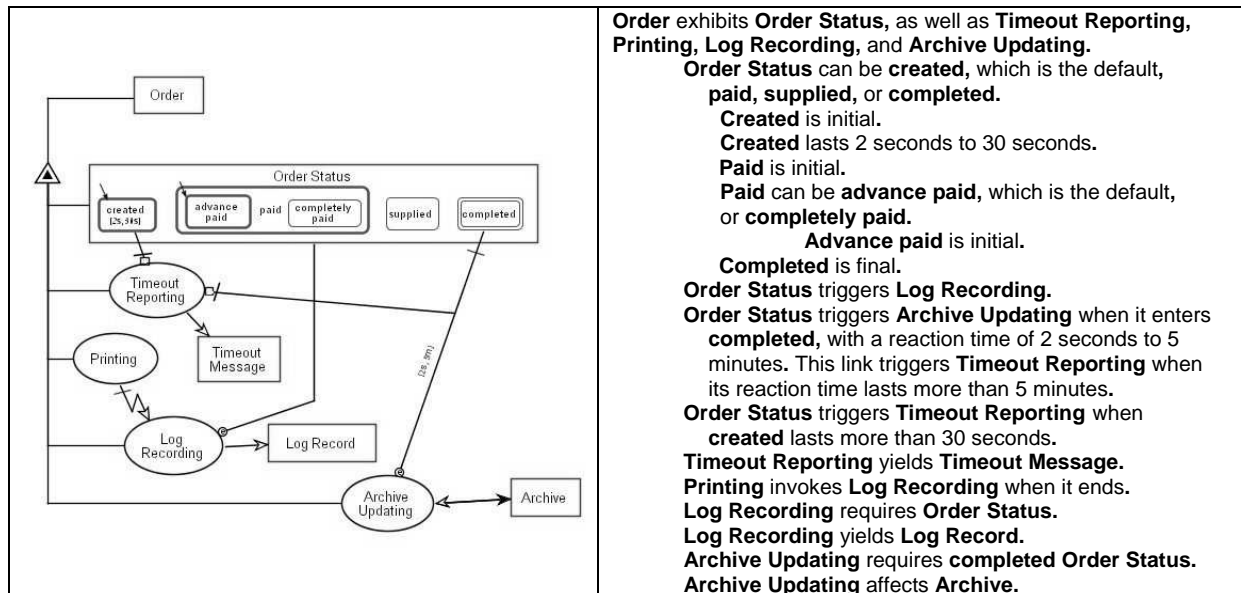


Figure 7. SD3, in which **Order** is unfolded, showing its operations and event triggers

There are five types of event links:

1. **Agent Link** – An *agent* is an intelligent object, a human or a group of humans, such as a department in an organization, who initiates a process by supplying an input signal (e.g., pushing a button or operating a machine) or supplying control data. An *agent link* is an event link which connects an agent with the process it triggers. The **Ordering** process in Figure 1 starts only when one of its agents, the physical and environmental (external) **Customer** or **Retailer**, enables its occurrence. The OPD symbol of an agent link is —● from the agent to the triggered process. In the OPL paragraph, this link is represented by the reserved word “handles”.
2. **State Change Event Links** – The fact that an object is at some state is a possible trigger for an event. In a *state change* event, the actual event can happen at any point in time between entrance to the state and exit from it. A state change event

link connects an object state with the process it triggers when entering or exiting the state. An enabling state change link is symbolized by $\text{---}\odot$, while a consumption state change link – by $\text{---}\blacktriangleright^e$.

A state change event has a timing attribute that determines at what point in time the event occurs along the stay of the object at the state. The possible values of the timing attribute are any, entrance, exit, and switch. The *any state change event* is an event that can occur at any point in time during the stay of the object at the state. The *state entrance event* occurs upon the object entering the state, while the *state exit event* means that the event occurs upon the object exiting (leaving) the state. The *state switch event* means that the event occurs upon the object either entering the state or exiting it. The timing of the event is denoted graphically by the *timing bar* – a small bar perpendicular to the event link, whose location along the link from the triggering state to the triggered process symbolizes the point in time at which the event occurs. Thus, an enabling state entrance event link is symbolized by $\text{+}\odot$, while a consumption state entrance event link is symbolized by $\text{+}\blacktriangleright^e$. An enabling state exit event link is symbolized by $\text{--}\odot$ and a consumption state exit event link is symbolized by $\text{--}\blacktriangleright^e$. Timing bars at both ends of the link denote a switch (entrance or exit) state event link, while no bar at all means a state change event link, where the event can take place at any point in time during the object's stay at the state.

In OPL, a triggering sentence is added to the OPL sentence representing the procedural aspect of the link. **Archive Updating** in Figure 7, for example, is triggered whenever **Order Status** enters its **completed** state. Two OPL sentences describe this link: the enabling sentence “**Archive Updating** requires **completed Order Status.**” and the triggering sentence “**Order Status** triggers **Archive Updating** when it

enters **completed.**” For a state exit event link, the OPL sentence would be “**Order Status** triggers **Archive Updating** when it exits **completed.**” For a state change event link which does not specify whether the event occurs upon entry to or exit from the state, the corresponding sentence would be “**Order Status** triggers **Archive Updating** when it is **completed.**” For a state switch event link, which specifies that the event occurs either upon entry to or upon exit from the state, the corresponding sentence would be “**Order Status** triggers **Archive Updating** when it either enters or exits **completed.**”

3. **General Event Links** – A general event can be an external stimulus, a change in an object state or value, etc. The source of a general event link is a thing (object or process). In Figure 7, for example, a general event link specifies a requirement that the **Log Recording** process is triggered any time **Order Status** changes its state. This single link could be replaced by five state entrance event links from each one of the bottom level states of **Order Status**, but the notation in Figure 7 is more compact. The **Log Recording** process does not change **Order Status**, as the enabling aspect (the circle) of the event link, $-\odot$, denotes. A general event link can also be of type consumption, symbolized by $-\blacktriangleright^*$, or effect, symbolized by \blacktriangleleft^* , denoting that the source object or process is respectively consumed or affected by the triggered process. The OPL sentence that specifies the triggering aspect of a general event link is "**Thing** triggers **Object.**" (for example, "**Order Status** triggers **Log Recording.**").
4. **Invocation Link** – An invocation link is a time-delimited event link between two processes – an invoking process and an invoked one. As noted, the vertical axis in an OPD denotes the time line within an in-zoomed process. The invocation link is used when this default process sequencing needs to be overridden, as in loops or jumping instructions. Using the timing bar symbol, an invocation link can trigger

the invoked process when the invoking process starts, denoted by $\overline{z\triangleright}$, ends, denoted by $\overline{z\triangleleft}$, starts or ends, represented by $\overline{z\triangleright}$, or at any time during its execution, represented by $\overline{z\triangleright}$. Figure 7 specifies that **Log Recording** is triggered any time **Printing** terminates. All the possible OPL invocation sentences are specified in Table 5 in Appendix B.

5. **Timeout Event Link** – A timeout event link is a time-delimited link that connects a timed element, which can be a process, a state, or an event link, with a process which is triggered when the element violates its time constraints. The timed element is constrained by minimal and/or maximal time limits. These constraints limit process execution, state duration, or the reaction time between triggering a process by an event link and the actual beginning of the triggered process. The timing bar denotes whether reference is made to the violation minimal, maximal, or either one of the two time constraints. When the timed element (timed process, timed state, or timed event link) violates its minimal time constraint, the minimal timeout event link, denoted by $\overline{+\square}$, is followed. When the element violates its maximal time constraint, the maximal timeout event link, denoted by $\overline{-\square}$, is followed. The symbol $\overline{+\square}$ represents a timeout event link which is followed whenever an extreme time constraints is violated, while $\overline{-\square}$ represents an unspecified timeout violation event. The square head of the timeout event link points towards the triggered process. The **created** state of **Order Status** in Figure 7, for example, is specified to last 2 to 30 seconds. If it lasts more than 30 seconds, it triggers the **Timeout Reporting** process, announcing the occurrence of a timeout error. All the possible OPL timeout sentences are specified in Table 5 in Appendix B.

As noted, an event link can have minimal and maximal reaction timeout constraints: if the triggered process does not start within the interval [minimal time constraint, maximal time constraint] after a stimulus occurred, a timeout event occurs. In Figure 7, for example, **Archive Updating** should be triggered within 2 seconds to 5 minutes after **Order Status** enters its **completed** state. If **Archive Updating** is not triggered within 5 minutes from that event, **Timeout Reporting** is triggered, announcing the reaction timeout error.

OPM REFLECTIVE METAMODEL

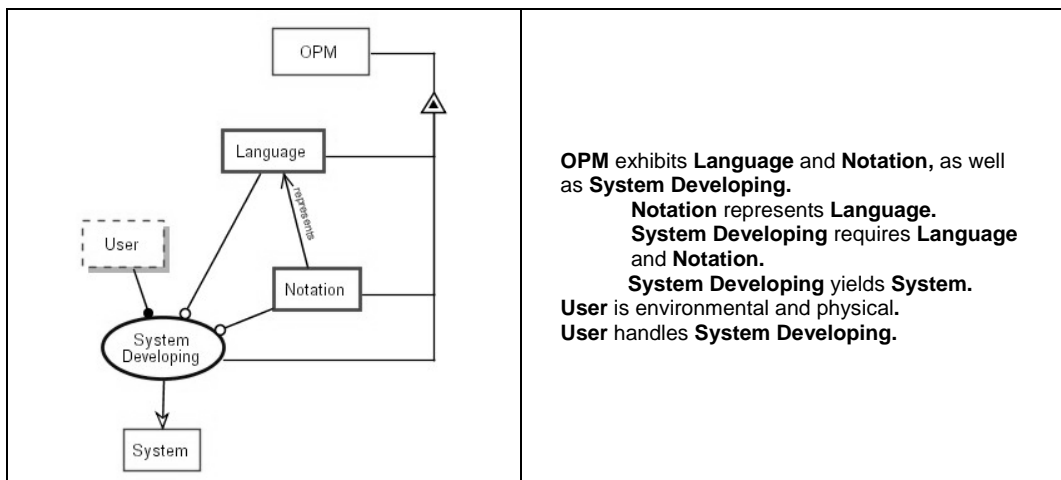
Up until now we have presented OPM in a rather informal way and accompanied the introduction with a running example. We devote the second part of this chapter to a formal reflective model of OPM. OPM is itself a complex system that combines language constructs and an approach to use that language. As such, it is amenable to modeling with any modeling language that is sufficiently expressive. In particular, it can be modeled in terms of OPM itself, yielding the OPM reflective metamodel. The rest of this chapter presents the language and notation parts of the OPM metamodel. As noted, the development part of OPM is the focus of (Dori and Reinhartz-Berger, 2003) and, hence, is not described here.

The Top Level Specification

The System Diagram (**SD**), which is the top-level, most abstract specification of the OPM metamodel, is presented in Figure 8. **SD** contains **OPM** and its features, which are the attributes **Language** and **Notation**, and the operation **System Developing**.

System Developing, which represents the entire OPM-based set of processes, is handled by the **User**, who is the agent of **System Developing**. This **User** can be the system architect, developer, or any other stakeholder who uses OPM to architect, develop, and evolve a **System**, as well as a team consisting of these stakeholders. The **System Developing** process

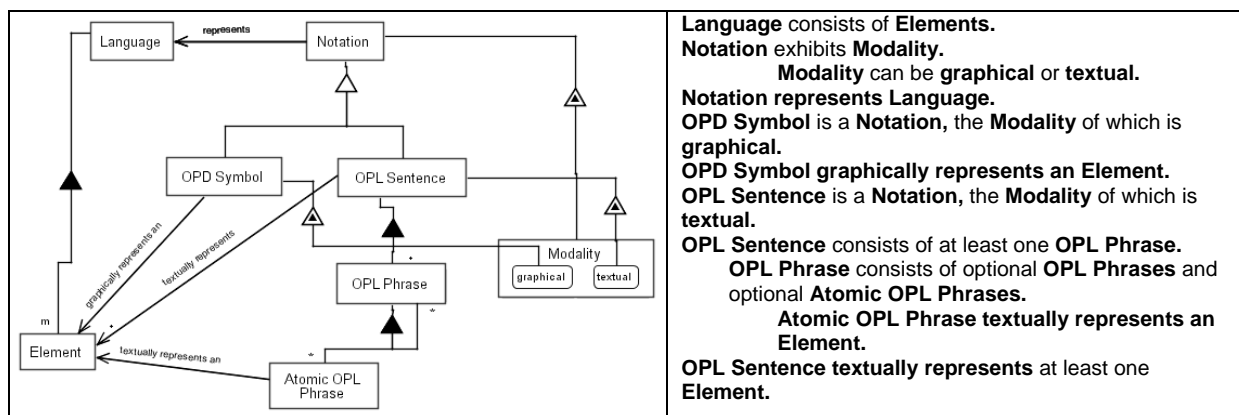
requires OPM's **Language** and **Notation** as instruments (unchangeable inputs) to create a new **System**.



OPM exhibits **Language** and **Notation**, as well as **System Developing**.
Notation represents **Language**.
System Developing requires **Language** and **Notation**.
System Developing yields **System**.
User is environmental and physical.
User handles **System Developing**.

Figure 8. **SD**, the top level specification, of the OPM reflective metamodel

OPM's **Language** encompasses OPM elements, their features, and the structural and procedural links among them, but it does not specify anything about the symbols used to denote them. The **Notation** represents the **Language** both visually, through interconnected OPD symbols, and textually, through OPL paragraphs and sentences.



Language consists of **Elements**.
Notation exhibits **Modality**.
Modality can be **graphical** or **textual**.
Notation represents **Language**.
OPD Symbol is a **Notation**, the **Modality** of which is **graphical**.
OPD Symbol graphically represents an **Element**.
OPL Sentence is a **Notation**, the **Modality** of which is **textual**.
OPL Sentence consists of at least one **OPL Phrase**.
OPL Phrase consists of optional **OPL Phrases** and optional **Atomic OPL Phrases**.
Atomic OPL Phrase textually represents an **Element**.
OPL Sentence textually represents at least one **Element**.

Figure 9. **SD1**, in which OPM **Notation** is unfolded

Unfolding **Notation**, **SD1** (shown in Figure 9) exposes the detailed relationships between **Language** and **Notation**. **Notation** is characterized by **Modality**, which has two possible states: **graphical** and **textual**. An **OPD Symbol** is a **Notation** the **Modality** of which is **graphical**, while an **OPL Sentence** is a **Notation** the **Modality** of which is **textual**. An **OPD Symbol** graphically represents an OPM **Element**, the building blocks of the **Language**, while an **OPL Sentence**

textually represents several **Elements**. An **OPL Sentence** may consist of several **OPL Phrases**, each of which can be an **Atomic OPL Phrase** or a complex **OPL Phrase**, i.e., one that consists of other **OPL Phrases**. An **Atomic OPL Phrase** **textually represents** a single **OPM Element**.

Element Metamodel

Figure 10 shows the third OPD of the OPM metamodel, labeled **SD2**, in which **Language** is unfolded. It specifies that **Language** consists of **Entities** and **Links**, each of which is an **Element**. An **Entity**, which exhibits (i.e., is characterized by) a **Name**, specializes into a **Thing** and a **State**. A **Thing** further specializes into an **Object** and a **Process**. The structural relation between an **Object** and a **State** represents that an **Object owns** some **States**, while a **State specifies the status of an Object**.

A **Link** exhibits **Homogeneity**, which is **homogeneous** for a **Structural Link** (that usually connects two **Objects** or two **Processes**) and **non-homogeneous** for a **Procedural Link** that usually connects an **Object** and a **Process**. The various types of links override this **Homogeneity** attribute when required.

Each **Element** is characterized by three orthogonal attributes:

- (1) **Affiliation**, which can be **systemic** (the default) or **environmental**. An **environmental Element** is an **Element**, the **Affiliation** of which is **environmental**. An **environmental Element** is external to the system or only partially specified, while a **systemic Element** is internal to the system and completely specified.
- (2) **Essence**, which can be **informatical** (the default) or **physical**. A **physical Element** consists of matter and/or energy. It can be a **physical Object** (e.g., a *Machine*), a **physical Process** (e.g., *Manufacturing*), a **physical State** (e.g., *tested*), or a **physical Link** (e.g., a communication line between two remote computers). An **informatical Element** relates to information.

(3) **Scope**, which can be **public** (the default), **protected**, or **private**. As in programming languages, the **Scope** of an **Element** can be **private** (i.e., it can be accessed only by itself), **protected** (accessible only by itself and its sub-elements), or **public** (accessible by any element in the system). Unlike the object-oriented paradigm, where a method can affect or access only the attributes of the same class, the default **Scope** in OPM is **public**, which implies that any OPM process can use or change all the objects in the model. While seemingly violating the object-oriented encapsulation principle, this provision increases the flexibility of modeling patterns of behavior as OPM processes that involve and cut across several object classes.

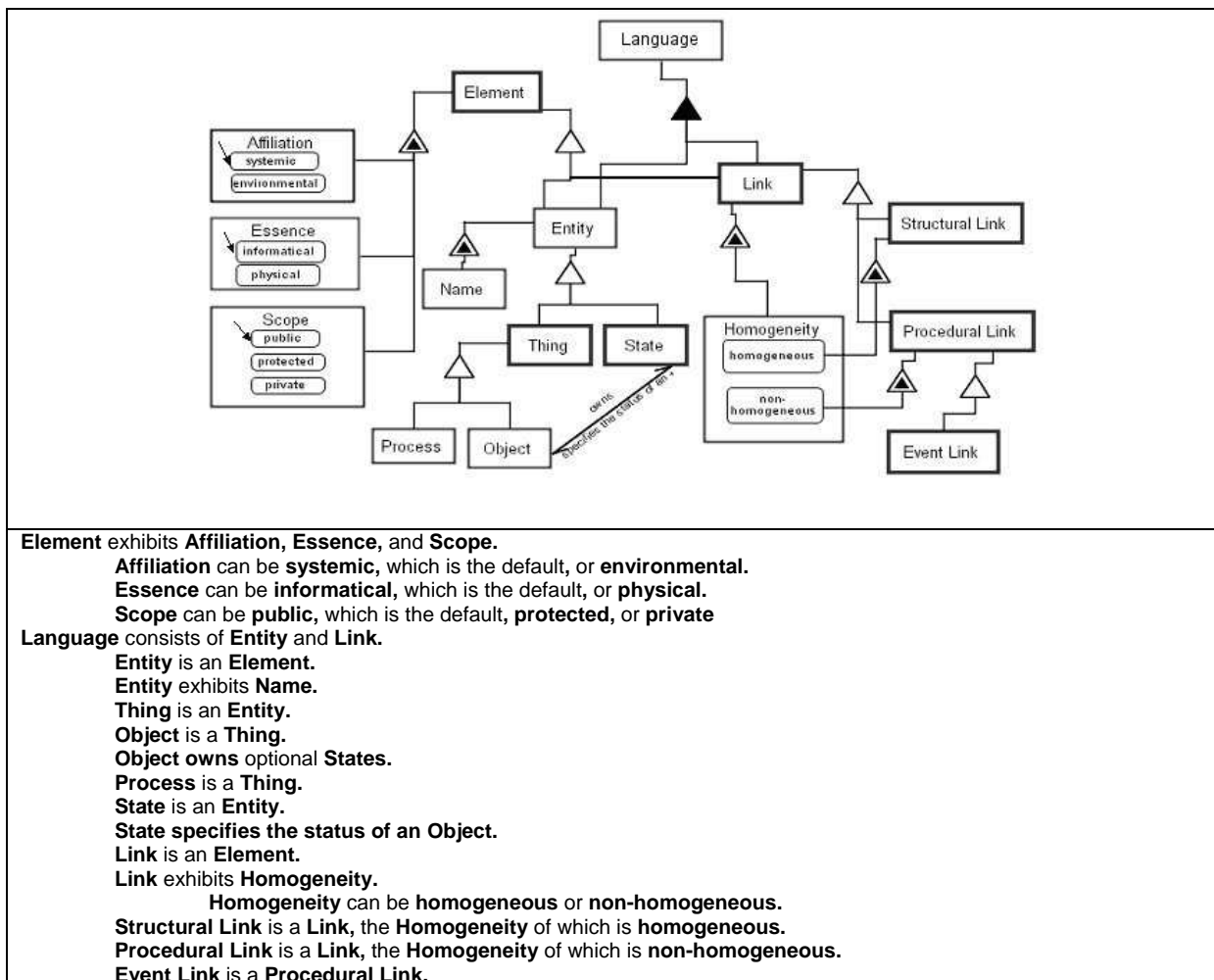
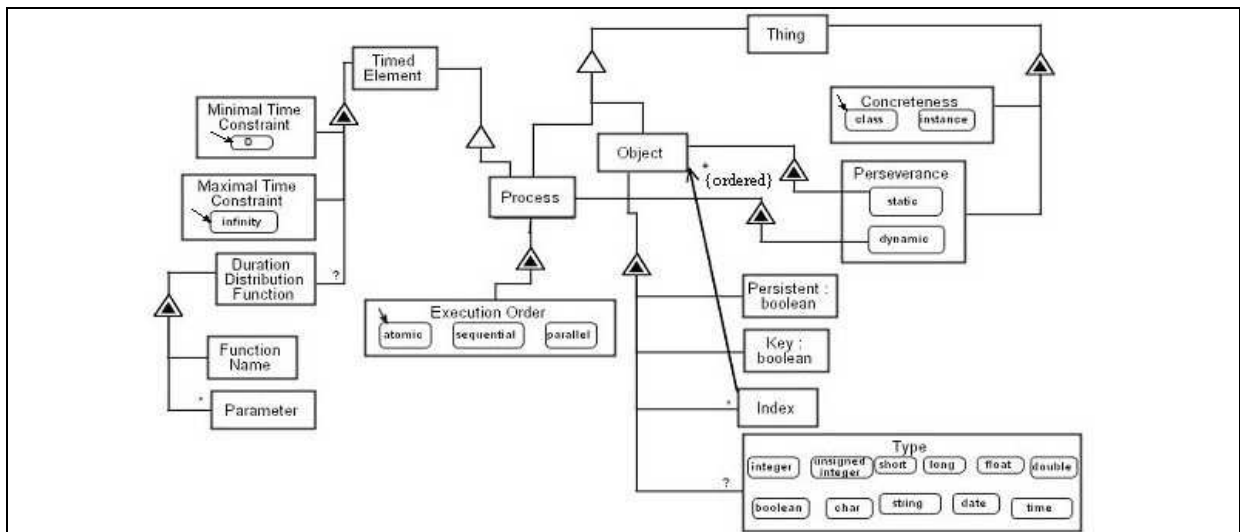


Figure 10. SD2, in which **Language** of OPM is unfolded

Thing Metamodel

Unfolding **Thing** of the OPM metamodel, **SD2.1** (Figure 11) shows its **Perseverance** attribute, which can be **static** or **dynamic**. An **Object** is a **Thing** with **static Perseverance**, while a **Process** is a **Thing** with **dynamic Perseverance**. In addition to **Perseverance**, a **Thing** also exhibits the **Concreteness** attribute, which determines whether the thing is a **class** (the default) or an **instance**. The difference between an **Object class** and an **Object instance** is similar to the difference between these concepts in the object-oriented approach. A **Process instance** is an occurrence of the process class, which, as noted, is a behavior pattern that the process instances follow. In programming terms, a **Process instance** can be thought of as an executable version of code, which can be executed a specified finite number of times, while a **Process class** is the complete code that can be (re)compiled and executed unboundedly. An **Object** can optionally exhibit **Type** (e.g., **integer**, **float**, or **string**), whether it is **Persistent** (i.e., stored in a database), whether it is **Key**, and optional **Indices**. Each **Index** is an ordered tuple of **Objects**.



Timed Element exhibits **Minimal Time Constraint**, **Maximal Time Constraint**, and an optional **Duration Distribution Function**.

Minimal Time Constraint is 0 by default.

Maximal Time Constraint is **infinity** by default.

Duration Distribution Function exhibits **Function Name** and optional **Parameters**.

Thing exhibits **Concreteness** and **Perseverance**.

Concreteness can be **class**, which is the default, or **instance**.

Perseverance can be **static** or **dynamic**.

Object is a **Thing**, the **Perseverance** of which is **static**.

Object exhibits **Persistent**, **Key**, optional **Indices**, and an optional **Type**.

Persistent is of type Boolean.

<p>Key is of type Boolean. Index relates to an ordered set of at least one Object. Type can be integer, unsigned integer, short, long, float, double, boolean, char, string, date, or time. Process is a Thing, the Perseverance of which is dynamic. Process is a Timed Element. Process exhibits Execution Order. Execution Order can be atomic, which is the default, sequential, or parallel.</p>
--

Figure 11. **SD2.1**, in which **Thing** of **OPM Language** is unfolded

Process, which is a **Thing** with a **dynamic Perseverance**, is also a **Timed Element** and as such it inherits **Minimal Time Constraint** (0 by default) and **Maximal Time Constraint** (infinity by default). As noted, these constraints limit the **Process** execution time within the specific bounds. **Process** also inherits from **Timed Element** a **Duration Distribution Function**, which is characterized by **Function Name** and **Parameters**. This function specifies the distribution of the process duration that determines how long a process execution lasts and it is most useful for simulation purposes.

In addition, **Process** exhibits **Execution Order**, which can be **atomic, sequential, or parallel**. Since a process can be either sequential or parallel (but not both), a zoomed-in process will have sub-processes that are all depicted either stacked or in a row, but not as a mixture of these two modes.

State Metamodel

A **State**, which describes a situation at which an **Object** can be, cannot stand alone, but is rather “owned” by an object. At any given point in time, an **Object** can be at exactly one of the **States** it owns, or in transition between two states. Like a **Process**, a **State** is a **Timed Element**, and as such it exhibits **Minimal Time Constraint** and **Maximal Time Constraint**, i.e., the minimal and maximal bounds for a continuous stay of the owning **Object** in that **State**. As a **Timed Element**, **State** also exhibits **Duration Distribution Function** for simulation purposes.

The OPD labeled **SD2.2** (Figure 12) specifies that a **State** has three additional Boolean attributes: **Initial**, **Final**, and **Default**. **Initial** determines whether the object can be initially (i.e., upon its creation) at this state. **Final** determines whether the object can be consumed

(destroyed) when it is at that state. **Default** determines whether this state is the default state (or value) of the owning object, i.e., the state into which the object enters when there is no specified initial state or more than one initial state. The self aggregation attached to **State** indicates that a state may recursively consist of lower-level **States**, which are nested sub-states.

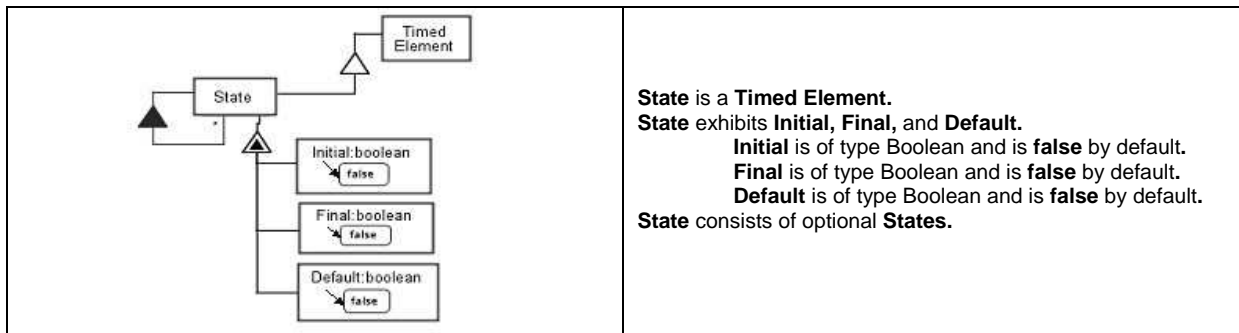


Figure 12. SD2.2, in which **State** of **OPM Language** is unfolded

Link Metamodel

As SD2.3 (Figure 13) shows, a **Link** exhibits two link ends: **Source End** and **Destination End**. Both are specializations of **Link End**, which is characterized by **Participation Constraint** (also known as multiplicity). **Participation Constraint** defines the **Minimal Cardinality** (with 1 as its default value) and the **Maximal Cardinality** (also 1 by default). These specify the minimal and maximal number of instances that can be connected by the link at the corresponding (source or destination) **Link End**. In addition a **Link** exhibits the **Homogeneity** attribute, which has two states: **homogeneous** and **non-homogeneous**. A **Link** is **homogeneous** if both its **Link Ends**, i.e., its **Source End** and **Destination End**, are linked to **Things** whose **Perseverance** value are the same. In other words, a **homogeneous Link** connects either two **Objects** or two **Processes**, while a **non-homogeneous Link** usually connects an **Object** to a **Process**. **Structural Links**, which denote static, non-temporal relations between the linked **Entities**, are usually **homogeneous Links**. **Procedural Links**, which model the behavior of the system along time

and represent flows of data, material, energy, or control between the linked entities, are **non-homogeneous Links** by default.

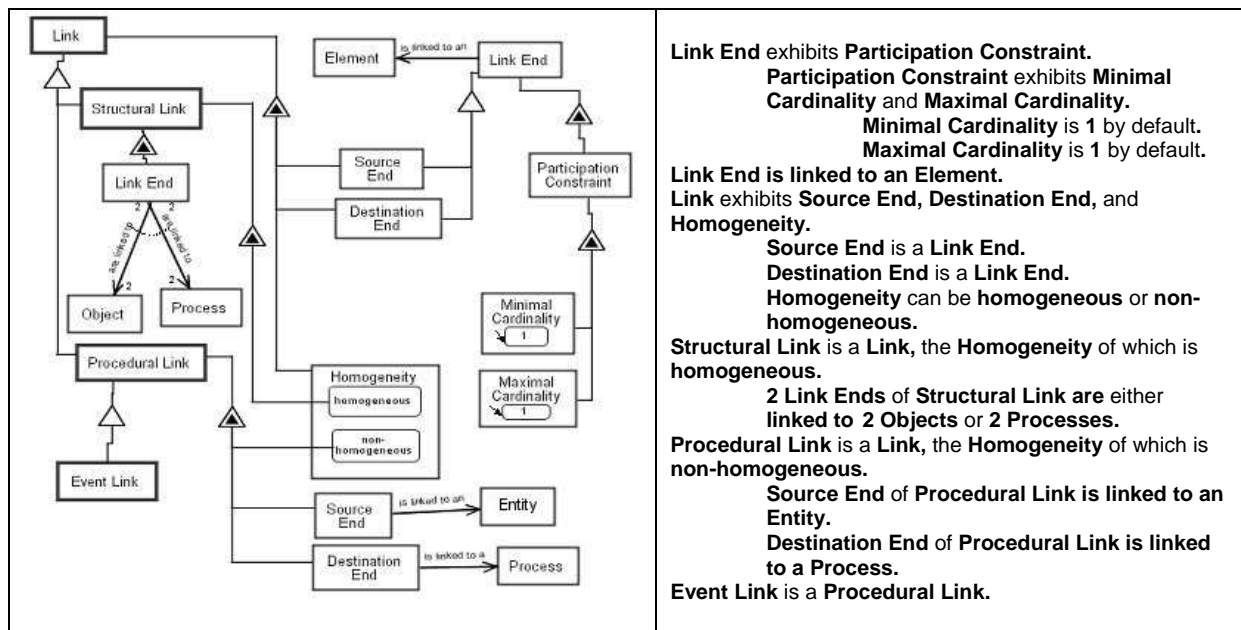


Figure 13. **SD2.3**, in which **Link** of **OPM Language** is unfolded

Determining link attribute values

The values of the **Essence**, **Affiliation**, and **Scope** link attributes, inherited from **Element**, are determined according to the corresponding values of the entities the link connects. If the entities have different values, a conflict arises that mandates a decision process based on three rules: the link essence, the link affiliation, and the link scope rules.

The **link essence rule** defines that the **Essence** value of a link is **physical** if the **Essence** of the two **Elements** it connects is **physical**. Hence, a **physical Link** can connect only two **physical Elements**, as described in Figure 14 by an OPM model.

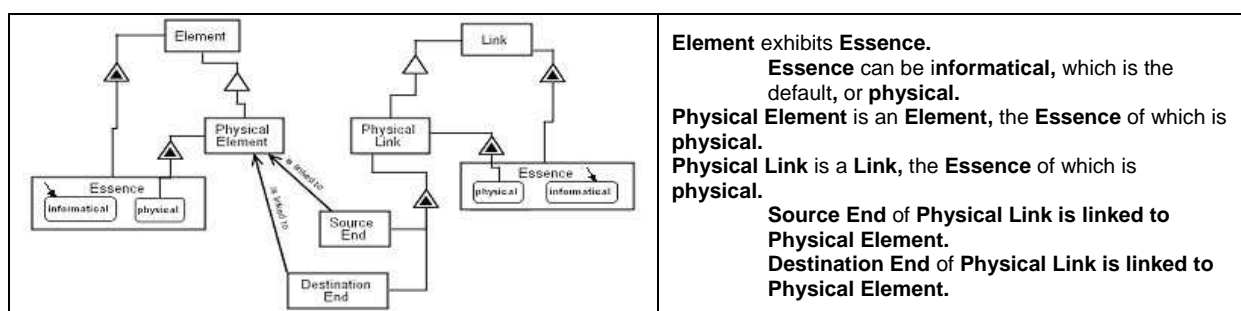


Figure 14. **SD2.3.1**, in which the **Link Essence** rule is specified

The *link affiliation rule* determines that the **Affiliation** value of a link is **environmental** if the **Affiliation** of the two **Elements** it connects is **environmental**. Hence, an **environmental Link** can connect only two **environmental Elements**, as specified in Figure 15.

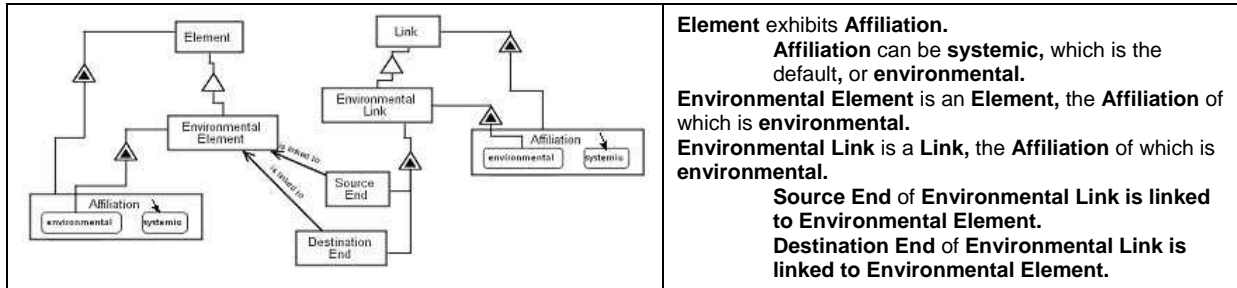


Figure 15. SD2.3.2, in which the **Link Affiliation** rule is specified

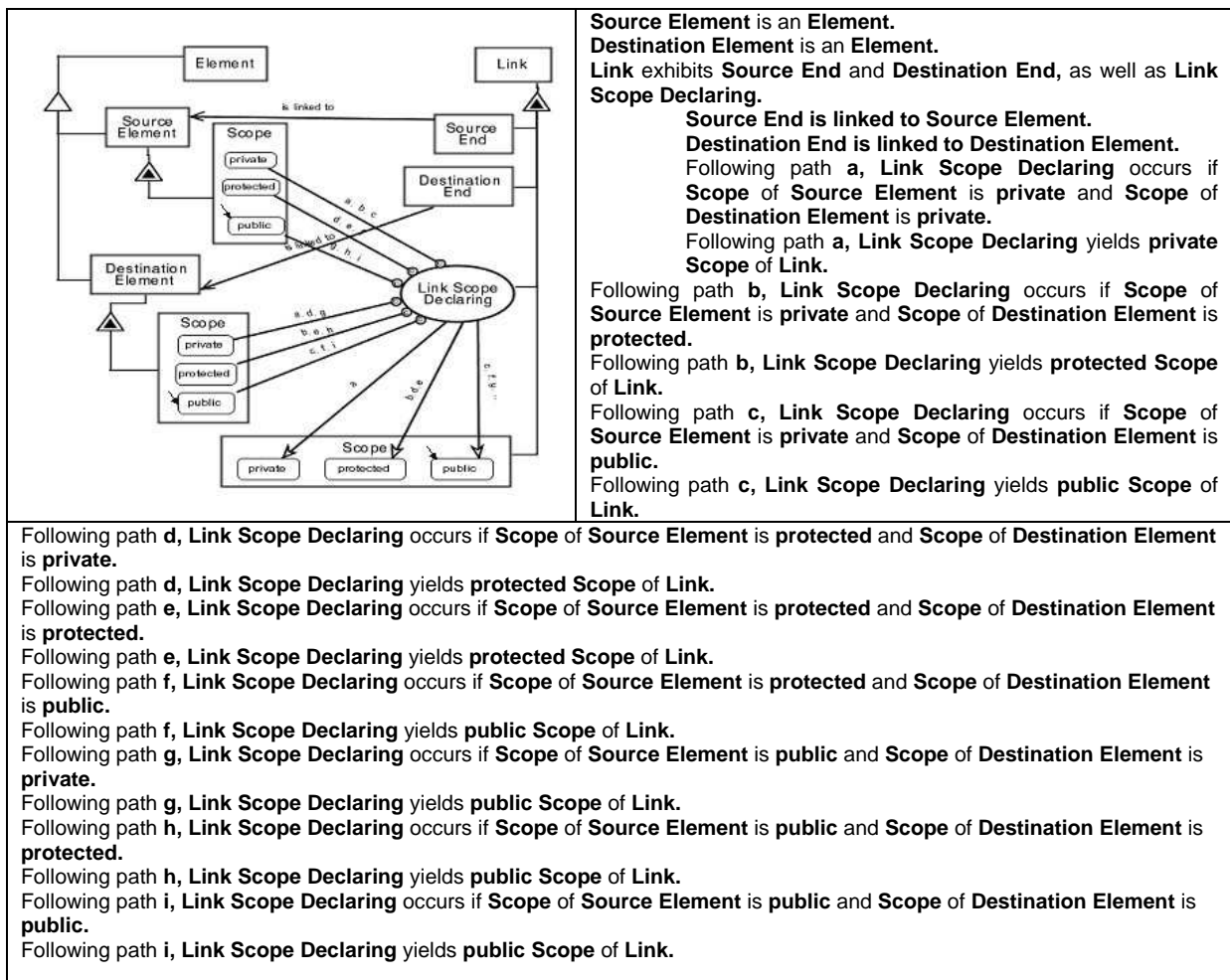


Figure 16. SD2.3.3, in which the **Link Scope** is specified

The *link scope rule* determines the **Scope** value of a **Link** as the widest of the **Scope** values of the two connected **Elements**, where **public**, **protected**, and **private** are the widest,

intermediate, and most narrow **Scope** values, respectively. Figure 16 specifies a process, **Link Scope Declaring**, that enforces this rule.

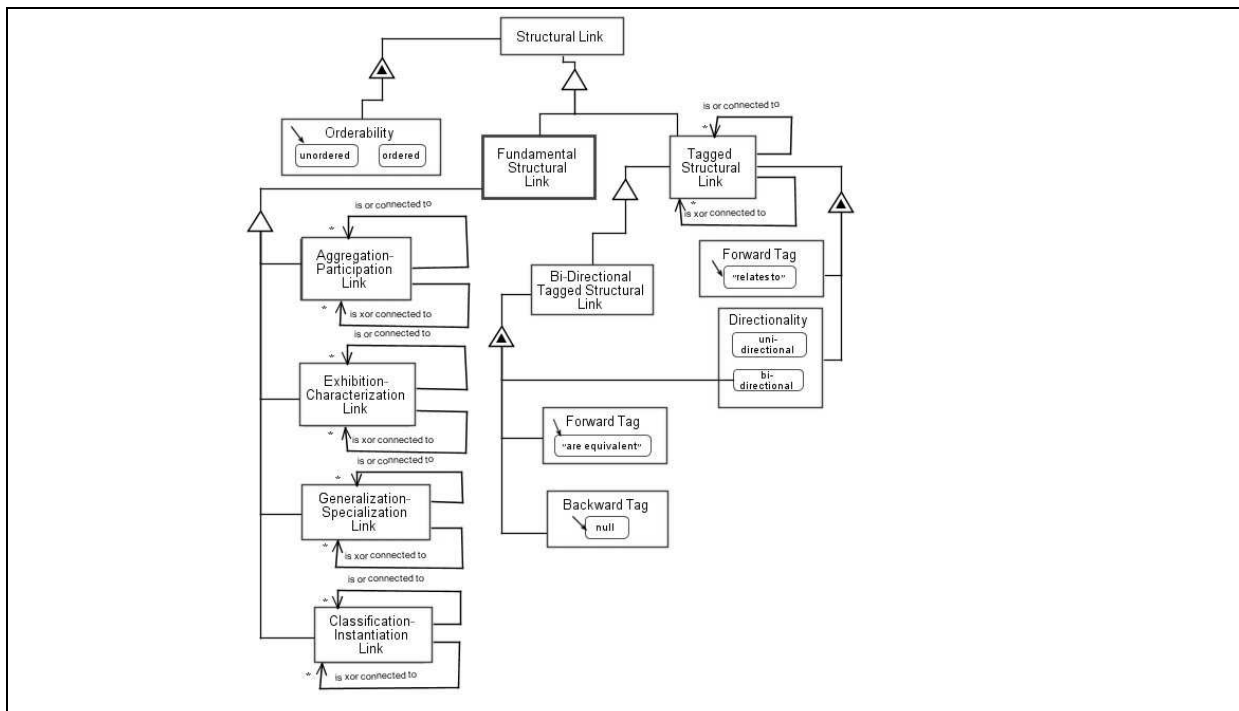
Structural Link Metamodel

SD2.4 (Figure 17) unfolds OPM **Structural Links**. A **Structural Link** is characterized by **Orderability**, which can be **ordered** (e.g., an array) or **unordered** (e.g., a set) by default. An **ordered Structural Link** adds the reserved label {ordered} next to the **Structural Link** symbol. In Figure 11, for example, **Object** is characterized by optional **Indices**, each of which is an ordered set of **Objects**.

SD2.4 also unfolds the two types of **Structural Links**: **Tagged Structural Links** and **Fundamental Structural Links**. A **Tagged Structural Link** exhibits **Forward Tag**, whose default value is the string “relates to”, and **Directionality**. A **Bi-Directional Tagged Structural Link**, which is a **Tagged Structural Link** whose **Directionality** is **bi-directional**, exhibits in addition **Backward Tag**, whose default value is **null**, and the default value of its (inherited) **Forward Tag** is “are equivalent”.

Fundamental Structural Links specialize into **Aggregation-Participation Link**, **Exhibition-Characterization Link**, **Generalization-Specialization Link**, and **Classification-Instantiation Link**. **Structural Links** of the same type can be connected by “OR” and/or “XOR” relations. This is specified by the self tagged structural links labeled “**is or-connected to**” and “**is xor-connected to**”, respectively.

SD2.4.1 (Figure 18), which unfolds the **Fundamental Structural Links**, specifies constraints on the **Elements** that can be connected by this type of links. Being **Structural Links**, **Fundamental Structural Links** connects two **Objects** or two **Processes**. There are two exceptions to this simple rule. These exceptions, which override the **Homogeneity** attribute of **Structural Links**, are explicitly specified in **SD2.4.1**:



Structural Link exhibits **Orderability**.
Orderability can be **unordered**, which is the default, or **ordered**.
Tagged Structural Link is a **Structural Link**.
Tagged Structural Link exhibits **Forward Tag** and **Directionality**.
Forward Tag is “relates to” by default.
Directionality can be **uni-directional** or **bi-directional**.
Tagged Structural Link is **xor-connected** to optional **Tagged Structural Links**.
Tagged Structural Link is **or-connected** to optional **Tagged Structural Links**.
Bi-Directional Tagged Structural Link is a **Tagged Structural Link**, the **Directionality** of which is **bi-directional**.
Bi-Directional Tagged Structural Link exhibits **Backward Tag**.
Backward Tag is **null** by default.
Forward Tag of **Bi-Directional Tagged Structural Link** is “are equivalent” by default.
Fundamental Structural Link is a **Structural Link**.
Aggregation-Participation Link is a **Fundamental Structural Link**.
Aggregation-Participation Link is **xor-connected** to optional **Aggregation-Participation Links**.
Aggregation-Participation Link is **or-connected** to optional **Aggregation-Participation Links**.
Exhibition-Characterization Link is a **Fundamental Structural Link**.
Exhibition-Characterization Link is **xor-connected** to optional **Exhibition-Characterization Links**.
Exhibition-Characterization Link is **or-connected** to optional **Exhibition-Characterization Links**.
Generalization-Specialization Link is a **Fundamental Structural Link**.
Generalization-Specialization Link is **xor-connected** to optional **Generalization-Specialization Links**.
Generalization-Specialization Link is **or-connected** to optional **Generalization-Specialization Links**.
Classification-Instantiation Link is a **Fundamental Structural Link**.
Classification-Instantiation Link is **xor-connected** to optional **Classification-Instantiation Links**.
Classification-Instantiation Link is **or-connected** to optional **Classification-Instantiation Links**.

Figure 17. SD2.4, in which **Structural Link** of **OPM Language** is unfolded

1. An **Exhibition-Characterization Link** connects a **Thing** or a **Link** (as its **Source End**) and an **Entity** (as its **Destination End**). For example, the communication link between remote computers, which is modeled as a **Tagged Structural Link**, can be characterized by the object *Transfer Rate* and/or the process *Encrypting*.
2. A **Generalization-Specialization Link** can connect two **States** of different **Objects** to represent state inheritance. In this type of link, which is called **State Generalization-**

Specialization Link, the inherited state has at least the same structural and procedural links as the inheriting state.

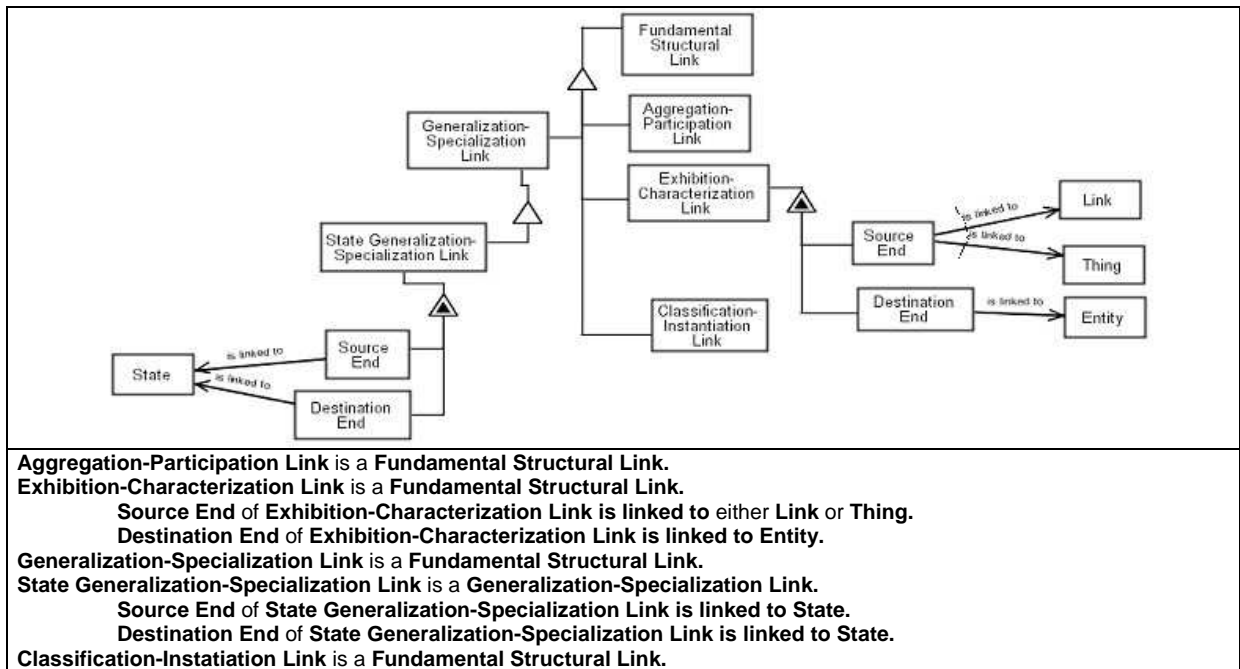


Figure 18. SD2.4.1, in which **Fundamental Structural Link** of **OPM Language** is unfolded

Table 1 summarizes the possible structural relations between OPM elements in a tabular way.

Table 1. Possible structural relations between OPM elements. S and D denote the link source and destination, respectively. + denotes a legal link.

Tagged Structural Link / Aggregation-Participation Link					Exhibition-Characterization Link				
D \ S	Object	Process	State	Link	D \ S	Object	Process	State	Link
Object	+	-	-	-	Object	+	+	-	+
Process	-	+	-	-	Process	+	+	-	+
State	-	-	-	-	State	+	+	-	+
Link	-	-	-	-	Link	-	-	-	-
Generalization-Specialization Link					Classification-Instantiation Link				
D \ S	Object	Process	State	Link	D \ S	Object	Process	State	Link
Object	+	-	-	-	Object	+	-	-	-
Process	-	+	-	-	Process	-	+	-	-
State	-	-	+	-	State	-	-	-	-
Link	-	-	-	-	Link	-	-	-	-

Procedural Link Metamodel

Any **Procedural Link** has a **Process** as its **Destination End**, while its **Source End** is connected to an **Entity**. As shown in **SD2.5** (Figure 19), a **Procedural Link** exhibits three attributes: **Link Type**, **Conditionality**, and optional **Path Labels**. The **Link Type** of a **Procedural Link** distinguishes primarily between **enabling** and **transforming Procedural Links**. **Transforming Procedural Links** are further divided into **affecting, consuming, and resulting Procedural Links**.

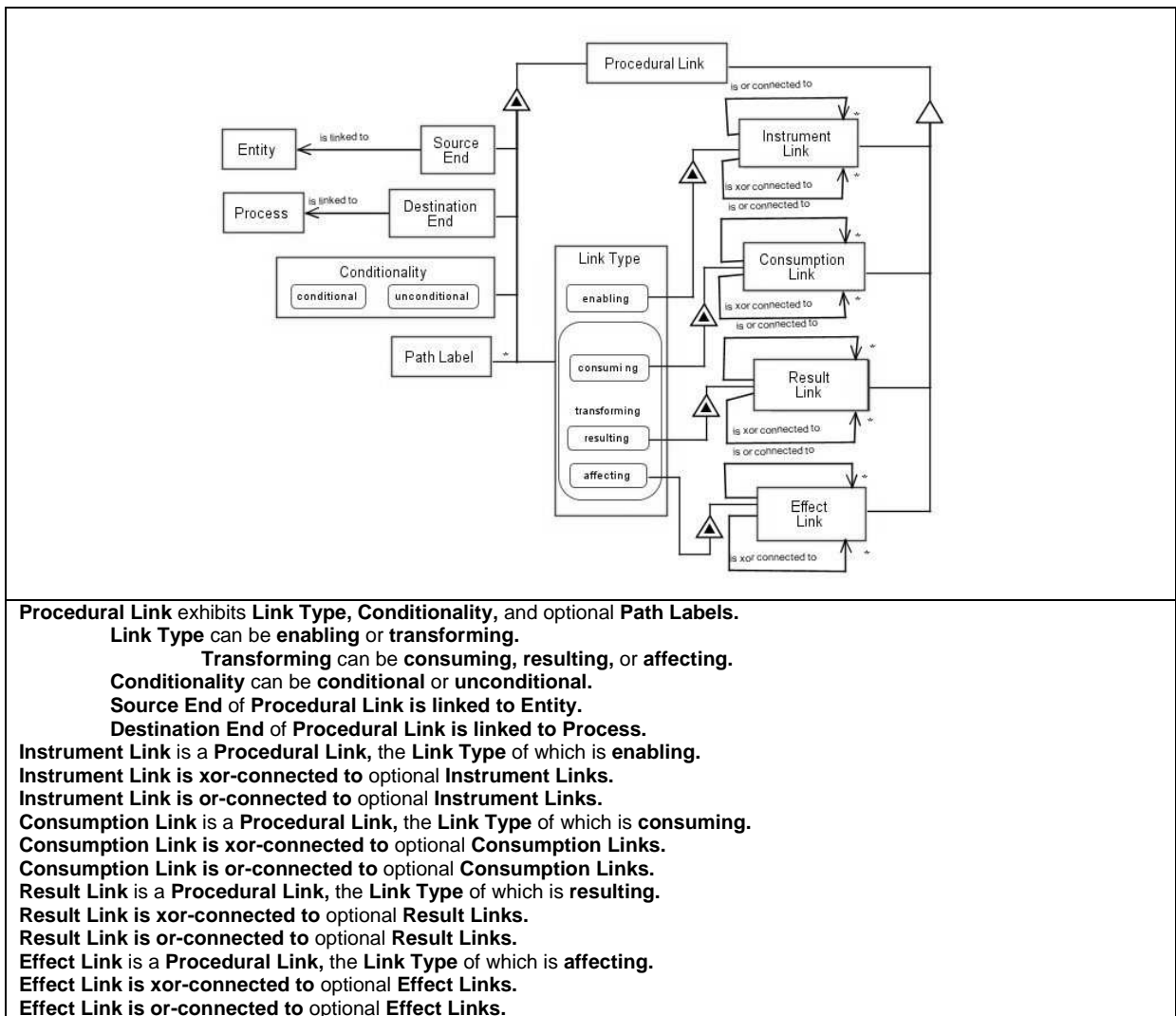


Figure 19. SD2.5, in which Procedural Link of OPM Language is unfolded

A **conditional Procedural Link**, i.e., a **Procedural Link** whose **Conditionality** is **conditional**, enables the **Process** execution only if the condition it symbolizes holds, else the destination **Process** is skipped and the next process in turn is examined for possible execution. With the

exception of **Result Link**, each type of procedural link can be either a **conditional Procedural Link** or an **unconditional Procedural Link**. A **Result Link** cannot be a **conditional Procedural Link** simply because the **Entity** which the **Process** generated upon its completion cannot be a condition for the **Process** that generated it.

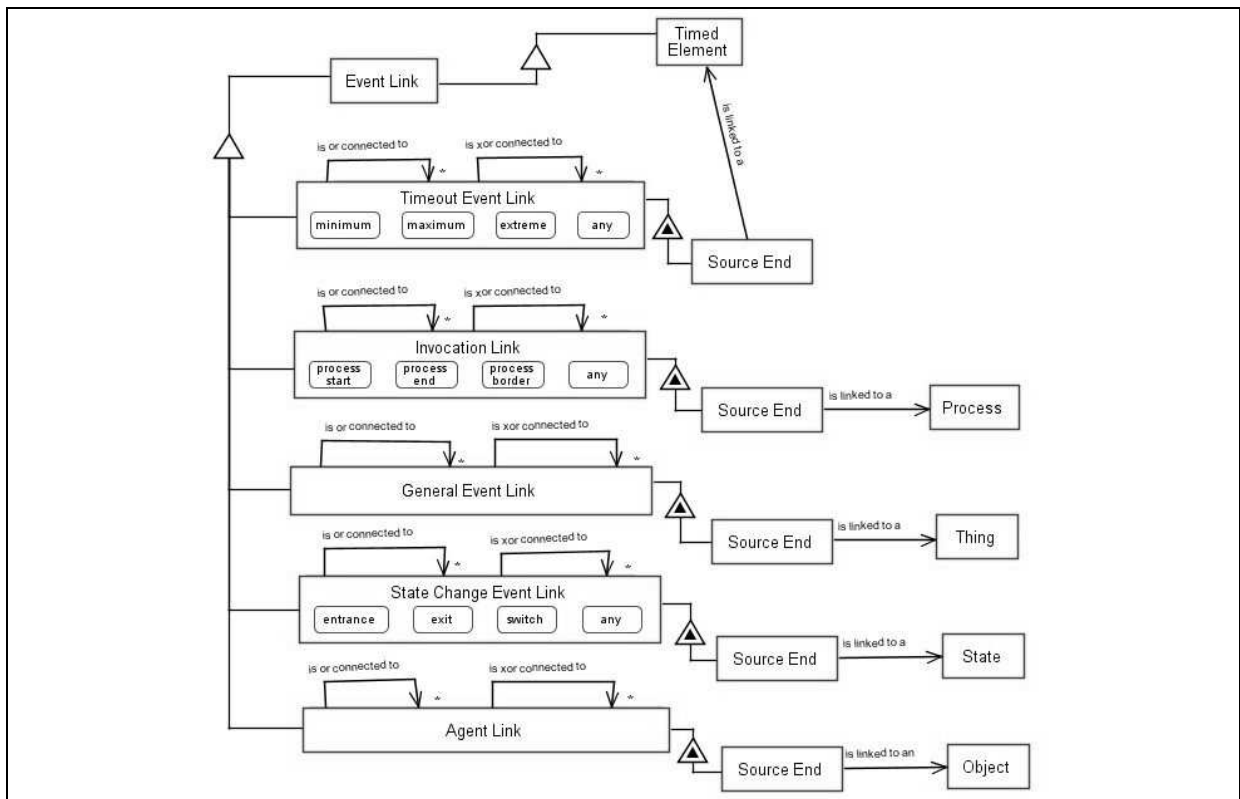
Like a **Structural Link**, a **Procedural Link** can be connected by “XOR” and “OR” relations to other **Procedural Links** of the same type, as shown by the self tagged structural links labeled “is xor-connected to” and “is or-connected to” in **SD2.5**.

Event Link Metamodel

As noted, an **Event Link**, which is unfolded in **SD2.6** (Figure 20), is a **Timed Element**. As such, it inherits **Minimal (reaction) Time Constraint**, **Maximal (reaction) Time Constraint**, and **Duration Distribution Function** as its attributes. The **Duration Distribution Function** of an **Event** can be used for system simulation to define the distribution of the time that passes from the event occurrence to the start of the corresponding triggered process.

SD2.6 also specifies the five types of **Event Links**: **Agent Link**; **State Change Event Link**, which can be **entrance State Change Event Link**, **exit State Change Event Link**, **switch State Change Event Link**, or **any State Change Event Link**; **General Event Link**; **Invocation Link**, which can be **process start Invocation Link**, **process end Invocation Link**, **process border Invocation Link**, or **any Invocation Link**; and **Timeout Event Link**, which can be **minimum Timeout Event Link**, **maximum Timeout Event Link**, **extreme Timeout Event Link**, or **any Timeout Event Link**.

An **Event Link** can be any **Procedural Link**, except for a **Result Link**, since the source **Entity** of a **Result Link** is created during the **Process** and, hence, cannot trigger it. An **Event Link** cannot be a conditional procedural link, since it triggers the process rather than just specifying an execution requirement on it.



Event Link is a Timed Element.
Timeout Event Link is an Event Link.
Timeout Event Link can be minimum, maximum, extreme, or any.
Source End of Timeout Event Link is linked to a Timed Element.
Timeout Event Link is xor-connected to optional Timeout Event Links.
Timeout Event Link is or-connected to optional Timeout Event Links.
Invocation Link is an Event Link.
Invocation Link can be process start, process end, process bordered, or any.
Source End of Invocation Link is linked to a Process.
Invocation Link is xor-connected to optional Invocation Links.
Invocation Link is or-connected to optional Invocation Links.
General Event Link is an Event Link.
Source End of General Event Link is linked to a Thing.
General Event Link is xor-connected to optional General Event Links.
General Event Link is or-connected to optional General Event Links.
State Change Event Link is an Event Link.
State Change Event Link can be entrance, exit, switch, or any.
Source End of State Change Event Link is linked to a State.
State Change Event Link is xor-connected to optional State Event Links.
State Change Event Link is or-connected to optional State Event Links.
Agent Link is an Event Link.
Source End of Agent Link is linked to an Object.
Agent Link is xor-connected to optional Agent Links.
Agent Link is or-connected to optional Agent Links.

Figure 20. SD2.6, in which Event Link of OPM Language is unfolded

COMPLEXITY MANAGEMENT IN OPM

As noted, OPM is a comprehensive systems evolution methodology. As such, it comprises not only a modeling language but also an approach for developing and evolving systems. Enabling both top-down and bottom-up development processes through its build-in complexity management mechanisms, OPM supports middle-out development. Complexity management aims at balancing the tradeoff between two conflicting requirements: completeness and clarity. Completeness requires that the system details be stipulated to the fullest extent possible, while the need for clarity imposes an upper limit on the level of complexity and does not allow for an OPD that is too cluttered or overloaded with entities and links among them. The seamless, recursive, and selective OPM scaling, i.e., refinement-abstraction, enables presenting the system at various detail levels without losing the “big picture” and the comprehension of the system as a whole.

Refinement-Abstraction Mechanisms

OPM features three built-in refinement-abstraction mechanisms, which are in-zooming and out-zooming, unfolding and folding, and state-expressing and state-suppressing.

In-zooming and out-zooming are a pair of refinement and abstraction mechanisms, respectively, which can be applied to all the three entity types: objects, processes, and states. In-zooming of (i.e., zooming into) an entity decreases the distance of viewing it, such that lower-level elements enclosed within the entity become visible. Conversely, out-zooming (i.e., zooming out) of a refined entity increases the distance of viewing it, such that all the lower-level elements that are enclosed within it become invisible. Figures 1, 3, 4, 5, and 6 are diagrams which result from in-zooming of different processes in the inventory system.

Unfolding/folding is a refinement/abstraction mechanism, which can be applied to things – objects or processes. *Unfolding* reveals a set of lower-level entities that are hierarchically

below a relatively higher-level thing. The hierarchy is with respect to one or more structural links. The result of unfolding is a graph the root of which is the thing being unfolded. Linked to the graph are the things that are exposed as a result of the unfolding. Conversely, *folding* is applied to a tree from which a set of unfolded entities is removed, leaving just the root. Figures 2 and 7 result from unfolding the order object of the inventory system. Unfolding/folding can be applied fully or partially to any subset of descendants (parts, specializations, features, or instances) of a thing (object or process).

State expressing is a refinement mechanism applied to objects which reveals a set of states inside an object. *State Suppressing* is the abstraction mechanism which conceals a set of states inside an object. For example, the order status in the inventory system is fully state-expressed in Figures 2 and 7 and only partially state-expressed in Figures 3, 4, 5, and 6. This object is state-suppressed in Figure 1.

Two entities in an OPD can be connected by at most one procedural link. While abstracting, a conflict between two competing links arises when an entity in the OPD is abstracted. A typical example is a process with two sub-processes, each of which is linked to the same object by a different procedural link, e.g. an instrument and a consumption link. When this process is out-zoomed, only one of these links needs to remain, and the question is which one prevails. The link needs to be at least as abstract as the more abstract link of the two competing links, so it may be one of these two procedural links or a third link which is more abstract than either one of them. In Figure 3, for example, the object **Order** is connected to the three sub-processes of **Ordering** through three links: a result link (to **Order Verification**) and two effect links (to **Customer Order Handling** and to **Retailer Order Handling**). When out-zooming of **Ordering**, the result link and the two effect links are replaced by a single result link, as shown in Figure 1. Figure 3 shows that **Order Status**, which is an **Order** attribute, is connected to **Receipt Generating** by two input (consumption) links and one output (result)

link. After suppressing the states of **Order Status**, this object remains connected to **Receipt Generating** with an effect link. Appendix C summarizes the abstraction order of procedural link by a table. This table defines for each two procedural links a third procedural link which replaces the two when abstracting (folding, out-zooming, or state-suppressing) the two procedural links. This table is the basis for defining the procedural aspects of OPM, which are also essential parts of the OPM reflective metamodel (Dori 2002, pp. 289-309; Dori and Reinhartz-Berger, 2003).

SUMMARY

A comprehensive reflective metamodel of OPM has been presented, using a bimodal representation of Object-Process Diagrams and Object-Process Language paragraphs. Although there seems to be a consensus among object-oriented languages that a system model should describe not just the structural aspect of a methodology but also its behavioral aspect (e.g., UML interaction diagrams), both the semantics and notations of system dynamics are confusing and incomplete. Furthermore, the metamodel of the UML industry standard depicts only the language part, leaving the (software or any other) system development processes informally as a “Unified Process.” In sharp contrast to this, OPM, being an object-process approach, enables reflective metamodeling of the complete methodology, including its language (with both its conceptual-semantic and notational-syntactic aspects) and the OPM-based system development process. This ability to create a reflective metamodel of OPM is indicative of OPM's expressive power, which goes hand in hand with OPM's ontological completeness according to the Bunge-Wand-Weber (BWW) evaluation framework (Soffer et al., 2001).

Besides being the source for OPM's definition, the reflective metamodel of OPM can serve other important goals. It can be used as a basis for a theoretical comparison between OPM and various object-oriented methods. COMMA, the Common Object-oriented Methodology

Metamodel Architecture, project (Henderson-Sellers and Bulthuis, 1998) used metamodeling to construct metamodels of popular object-oriented methodologies and identify a core that was later used as a basis for OPEN, Object Process, Environment, and Notation (OPEN site, 2003). The OPM metamodel can be compared to these metamodels and an automatic transformation generator can be made between popular object-oriented methodologies, such as UML, and OPM. Indeed, OPCAT, Object-Process CASE Tool, (Dori et al., 2003) can automatically generate a set of UML views, including use case, class, sequence, activity, Statecharts, and deployment diagrams, from the single OPM model.

The reflective OPM metamodel helps also define an implementation generator, which automatically transforms the OPM model resulting from the system's analysis and design into a database scheme and executable code. The benefits of this implementation generation include increasing productivity and quality; enabling mechanical and repetitive operations to be done quickly, reliably and uniformly; and relieving designers from mundane tasks so they can focus on creative tasks that require human intelligence. OPM-GCG (Reinhartz-Berger and Dori, 2004), the generic code generator of OPM, handles dynamic repositories of translation rules from an XML syntax of Object-Process Language to various target programming languages. These translation rules are defined in a separate offline tool and are used by the implementation generator at will. Being based on OPM, OPM-GCG enables the generation of potentially complete application logic rather than just skeleton code.

The different OPM system development and evolution processes, as well as the refinement and abstraction mechanisms, provide a theoretical foundation for improving OPCAT to make it a fully Integrated System Engineering Environment (I SEE). OPCAT already supports system simulation during the design phase, OPD generation from an OPL script, OPL generation from an OPD-set, and implementation generation.

REFERENCES

- Clark, T., Evans, A., & Kent, S. (2002). Engineering Modeling Languages: a Precise Meta-Modeling Approach. 5th International Conference on Fundamental Approaches to Software Engineering (FASE'2002), 159-173.
- Dori, D. (2002). [Object-Process Methodology - A Holistic Systems Paradigm](#). Springer Verlag Press.
- Dori, D., & Reinhartz-Berger, I. (2003). Reflective Metamodel of OPM – An OPM-Based System Development Process. Proceedings of the 22nd International Conference on Conceptual Modeling (ER'2003), Lecture Notes in Computer Science 2813, pp. 105-117.
- Dori, D., Reinhartz-Berger, I., & Sturm A. (2003). OPCAT – A Bimodal Case Tool for Object-Process Based System Development. Proceedings IEEE/ACM 5th International Conference on Enterprise Information Systems (ICEIS 2003), 286-291.
- Download site of the software: <http://www.objectprocess.org/>
- Harel, D. (1987). Statecharts: a Visual Formalism for Complex Systems, Science of Computer Programming, 8, 231-274.
- Henderson-Sellers, B., & Bulthuis, A. (1998). Object-Oriented Metamethods. New York: Springer Verlag Press.
- Mayer, R.E. (2001). Multimedia Learning. New York: Cambridge University Press.
- Metamodel site. (2003). What is metamodelling, and what is a metamodel good for? <http://www.metamodel.com/>
- Nuseibeh, B., Finkelstein, A., & Kramer, J. (1996). Method engineering for multi-perspective software development. Information and Software Technology journal, 38 (4), 267-272.
- Object Management Group (OMG). (2001) UML 1.4 - UML Semantics. OMG document formal/01-09-73, <http://cgi.omg.org/docs/formal/01-09-73.pdf>
- Object Management Group (OMG). (2003). Meta Object Facility (MOF) Specification. OMG document formal/02-04-03, <http://cgi.omg.org/docs/formal/02-04-03.pdf>
- OPEN web site. (2003). <http://www.open.org.au/>
- Peleg, M., & Dori, D. (1999). Extending the Object-Process Methodology to Handle Real-Time Systems, Journal of object-oriented programming, 11 (8), 53-58.
- Peleg, M. & Dori, D. (2000). The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. IEEE Transaction on Software Engineering, 26 (8), 742-759.
- Reinhartz-Berger, I., & Dori, D. (2004). Object-Process Methodology (OPM) vs. UML: A Code Generation Perspective. Accepted to the 9th CAiSE/IFIP8.1/EUNO International

- Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'04).
- Reinhartz-Berger, I., & Dori, D. (2004). OPM vs. UML – Experimenting Comprehension and Construction of Web Application Models. Accepted to Empirical Software Engineering journal.
- Reinhartz-Berger, I., Dori, D., & Katz S. (2002). OPM/Web - Object-Process Methodology for Developing Web Applications. Annals of Software Engineering – Special Issue on Object-Oriented Web-based Software Engineering, 141–161.
- Reinhartz-Berger, I., Dori, D., & Katz S. (2002). Open Reuse of Component Designs in OPM/Web. Proceeding of Computer Software and Application Conference (COMPSAC'2002), 19-24.
- Rosemann, M., & Green, P. (2002). Developing a Meta Model for the Bunge-Wand-Weber Ontological Constructs. Information Systems, 27, 75-91.
- Rossi, M., Tolvanen, J.P., Ramesh, B., Lyytinen, K., & Kaipala, J. (2000). Method Rationale in Method Engineering. Proceedings of the 33rd Hawaii International Conference on System Sciences, <http://www.computer.org/proceedings/hicss/0493/04932/04932036.pdf>
- Soffer, P., Golany, B., Dori, D., & Wand, Y. (2001). Modelling Off-the-Shelf Information Systems Requirements: An Ontological Approach. Requirements Engineering, 6 (3), 183-199.
- Soffer, P., Golany, B., & Dori, D. (2003). ERP Modeling: A Comprehensive Approach. Information Systems, 28(6), 673-690.
- Van Gigch, J. P. (1991). System Design Modeling and Metamodeling. Kluwer Academic Publishers.
- Warmer, J. & Kleppe, A. (1999). The Object Constraint Language – Precise Modeling with UML. Addison-Wesley.
- Wand, Y. & Weber, R. (1993). On the Ontological Expressiveness of Information Systems Analysis and Design Grammars. Journal of Information Systems, 3, pp. 217-237.

APPENDIX A. BWW ONTOLOGICAL CONSTRUCTS AND THEIR OPM REPRESENTATION

Table 2. BWW ontological constructs and their mapping to OPM concepts

Ontological Construct	BWW Explanation	OPM Representation
Thing	A thing is the elementary unit in the ontological model. The real world is made up of things. A composite thing may be made up of other things	An instance
Property	Things possess properties. A property is modeled via an attribute function that maps the thing into some value	An attribute is an object related to another object by a characterization link
Class	A class is a set of things that possess common properties	An object class
State	The vector of values for all attribute functions of a thing is the state of the thing	A state (separately modeled for each attribute)
State law	A state law restricts the values of the properties of a thing to a subset defined by natural or human laws	A state law is a specification of the possible states of an object, including distinction of transient and persistent states
Event	An event is a change of state of a thing, effected via a transformation (see below)	The event of changing state A to state B is represented by the sequence <State A → consumption link → process → result link → state B>
Transformation	A transformation is a mapping from one state to another one	A process (class)
Lawful transformation	A lawful transformation defines which events in a thing are lawful	A set of objects / states linked to a process by a condition / event / effect / consumption / instrument link. The process is linked to another set of objects / states by an effect / result link
External event	An event that arises in a thing, subsystem or system by virtue of the action of some thing in the environment on the thing, subsystem or system	Object / state → event link → process
Internal event	An event that arises in a thing, subsystem or system by virtue of lawful transformations in the thing	Process → effect / result link → object / state
Stable State	A state in which a thing, subsystem or a system will remain unless forced to change by virtue of the action of a thing in the environment (an external event)	A persistent state, or any other state, which is not unstable (see below)
Unstable state	A state that will be changed into another state by virtue of the action of transformations in the system	State A in the sequence <state A → condition / event / consumption link → process → result link → state B> is an unstable state
Subclass	A subset of a class, defined by a conjunction of properties	An object class, which is related to another class by a specialization link
Composition	The things in a composite thing are its composition	Composition and decomposition are given by the sequence <object → aggregation link → set of objects>. The composite thing is linked at the vertex of the aggregation symbol and its components at the bottom
Decomposition	A decomposition of a composite thing is a set of things such that every component of the composite thing is either a member of this set or is included in the composition of one of the members	

APPENDIX B. OPM CONCEPTS AND SYMBOLS

Table 3. Entities – Things and States













	Entity Type	Entity Symbol
Object	Systemic, informatical object	
	Environmental, informatical object	
	Systemic, physical object	
	Environmental, physical object	
Process	Systemic, informatical process	
	Environmental, informatical process	
	Systemic, physical process	
	Environmental, physical process	
State	Regular state	
	Initial state	
	Final state	
	Default state	

Table 4. Structural Relations, their OPD symbols, and OPL sentences



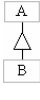

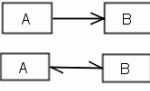


Structural Relation Name	OPD Symbol	OPL Sentence
Aggregation-Participation		A consists of B.
Exhibition-Characterization		A exhibits B.
Generalization-Specialization		B is an A.
Classification-Instantiation		B is an instance of A.
Tagged Structural Link		A relates to B. A and B are equivalent.
XOR relation		E.g., A relates to either B or C.
OR relation		E.g., A relates to B or C.

Table 5. Procedural Links, their OPD symbols, and OPL sentences

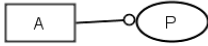
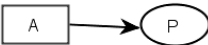
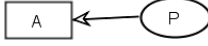

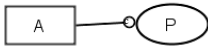
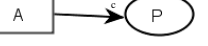
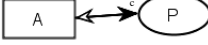


Type	Link Name	Semantics	OPD Symbol	OPL Sentence
Enabling Links	Instrument	The process requires the entity, but does not change it during execution.		P requires A.
Transforming Links	Consumption	The process consumes the entity.		P consumes A.
	Result	The process generates (creates) the entity.		P yields A.
	Effect	The process changes (affects) the thing.		P affects A.
Conditional Links	Instrument	The process occurs if the entity exists (in some state). The process requires the entity.		P occurs if A exists. P requires A.
	Consumption	The process occurs if the entity exists (in some state). The process consumes the entity.		P occurs if A exists. P consumes A.
	Effect	The process occurs if the thing exists. The process changes (affects) the thing.		P occurs if A exists. P affects A.
Logical Relations	XOR relation			E.g., P affects either A or B.
	OR relation			E.g., P affects A or B.

Table 6. Event links, their semantics and symbols

Event Type	Semantics	OPD Symbol	OPL Sentence
Agent	The process is triggered by the intelligent object.		A handles P.
State Change	The process is triggered when the object enters or exits the state. The object may be changed.	Enter: Exit: Switch: Any:	A triggers P when it enters/exists/either enters or exists st . St A triggers P.
General Event	The process is triggered when the object or process is changed or cause external stimuli. The object may be consumed or changed.		A triggers P.
Invocation	The process is triggered when the source process starts or ends.	Start: End: Border: Any:	P invokes P1 when it starts/ends/ either starts or ends. P invokes P1.
Minimal or Maximal State Timeout	The process is triggered when the object violates its minimal or maximal time constraints for staying at the state.	Min: Max: Extreme: Any:	A triggers P when st lasts less than Time / more than Time /less than Time or more than Time . Timeout of st A triggers P .
Minimal or Maximal Process Timeout	The process is triggered when the process violates its minimal or maximal execution time constraints.	Min: Max: Extreme: Any:	P1 triggers P when it lasts less than Time / more than Time / either less than Time or more than Time . Timeout of P1 triggers P .
Reaction Timeout	The process is triggered when the event link violates its minimal or maximal reaction time constraints.	Min: Max: Extreme: Any:	This link triggers P when its reaction time lasts less than Time / more than Time / either less than Time or more than Time . This link timeout triggers P .
XOR relation			E.g., A triggers either P or Q when it changes.
OR relation			E.g., A triggers P or Q when it changes.

Comment: The OPL sentences in this table are for the event aspect of the link. For state change and general event links, an additional OPL sentence, which represents its procedural aspect, should be added.

