

A Graph Grammar-Based Formal Validation of an Object-Process  
Diagram

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of  
Master of Science in Information Management Engineering

Arieh Bibliowicz

Submitted to the Senate of  
the Technion – Israel Institute of Technology

Av, 5768 Haifa August 2008

The Research Thesis was done under the supervision of Professor Dov Dori in the faculty of Industrial Engineering, Information Management Area.

## Table of Contents

1. Introduction and Background	1
2. Object-Process Methodology	8
3. Graph Grammars	16
4. OPD Graph Grammar	24
5. OPD Abstraction	32
6. Coverage	86
7. From a single OPD to a Complete OPM System Validation	88
8. Conclusions	93
9. References	94

## Table of Figures

Figure 1. The OPM entities	9
Figure 2. Example of OPM links	10
Figure 3. OPD hierarchy example	14
Figure 4. Basic graph	16
Figure 5. The two productions of Example 1	19
Figure 6. Initial graph of Example 1	20
Figure 7. Example 1 graph after application of Production 1 – <i>Consumption Link Insertion</i>	21
Figure 8. Example 1 graph after application of <i>Consumption Link Insertion</i> and <i>Object Aggregation Removal</i>	21
Figure 9. Application Conditions	22
Figure 10. Production 1 with a negative constraint	23
Figure 11. Abstract links	25
Figure 12. Abstract structural and procedural links	25
Figure 13. Thing Creation production	26
Figure 14. Existing Name Thing Creation Production	26
Figure 15. State Creation production	27
Figure 16. State Removal production	27
Figure 17. Thing Removal production	27
Figure 18. Homogeneous structural link creation productions	28
Figure 19. Aggregation loop creation production	29
Figure 20. Generalization and Aggregation link pair creation production	29
Figure 21. Non-homogeneous structural link creation production	29
Figure 22. Object-to-Process link creation productions	30
Figure 23. Process-to-Object link creation production	30
Figure 24. Bidirectional procedural link creation production	31
Figure 25. Invocation link creation production	31
Figure 26. Link removal production	31
Figure 27. Abstraction of a simple OPM link	32

## Table of Figures (Cont.)

Figure 28. Abstraction of the link of a part	32
Figure 29. Temporary instrument link (left) and result link (right)	33
Figure 30. Superpositioned temporary and regular links together	34
Figure 31. Calculating the things' height	36
Figure 32. State change abstraction production	37
Figure 33. State-Specified Link Abstraction production	37
Figure 34. Assembly Line OPD before abstraction	38
Figure 35. OPD after abstraction	38
Figure 36. Production 9.2.1 – Promotion of Part Effect to Aggregate Effect	39
Figure 37. Match in Assembly Line OPD for Production 9.2.1	39
Figure 38. Object exhibiting process	67
Figure 39. Example embedding of Parent OR/XOR production in an Object-Based Procedural Link Abstraction production.	68
Figure 40. Illegal construct example	70
Figure 41. Invalid signature example	77
Figure 42. Valid signature abstraction	78
Figure 43. Example OPD before and after abstraction	78
Figure 44. Abstraction process example execution	79
Figure 45. ABS Braking OPD	81
Figure 46. State change abstraction on Brake Assembly	82
Figure 47. State-Specified Link abstraction on Brake Assembly	82
Figure 48. Example diagram after initial abstraction of Brake Assembly	82
Figure 49. Promotion of Part Instrument to Aggregate Instrument Production	83
Figure 50. Example diagram after Brake Assembly instrument link abstraction	83
Figure 51. Example diagram after all Brake Assembly links abstracted	83
Figure 52. Example diagram after first round of the abstraction algorithm	84
Figure 53. Example diagram before Braking process abstraction	84
Figure 54. Example diagram after procedural abstraction of Braking process	85
Figure 55. Example diagram after abstraction of Braking process	85
Figure 56. Final abstraction of the example diagram	85
Figure 57. In-Zoom Refinement	89

## Table of Figures (Cont.)

Figure 58. In-Zoom Refinement Unfolding Transformation	89
Figure 59. The link precedence matrix	90

## Abstract

Conceptual modeling is the field where humans model natural or artificial systems. The basic requirements of a model is for it to be both easy to understand and to describe correctly and unambiguously the system that it is modeling. Although these goals are trivial, over time most modeling methodologies have been able to satisfy only one of the requirements fully, regularly the ease to understand, while leaving the correctness and ambiguousness problems aside. There exist some formal modeling languages but a full model is fairly complicated to understand in these languages.

OPM is an holistic system modeling methodology that combines all of the aspects of a system in one model, and at the same time provides mechanisms to manage the complexity of the model with zooming and folding operations, which divide the complexity of a large system into many models that are interconnected. Although the basic syntax and semantics of an OPM system model are already defined, this definitions is not complete and leaves room for incorrect models and models that can be interpreted in different ways.

This work advances de formal definition of OPM by providing a graph grammar which creates OPM diagrams whose syntax is correct when a pair of interconnected things are taken, but the correctness of the whole diagram is not assured. After that a diagram is created, a validation methodology must be applied to the diagram to assure that it is fully semantically and syntactically correct. The validation methodology is also based on graph grammars.

# 1. Introduction and Background

Conceptual modeling is the field where humans model systems. The various uses of these models can be divided into two groups: models that describe existing things (for example the interchange of chemicals in a human cell) and models that describe human creations (such as machines, software, assembly lines). We want these models to be simple, abstracting information that is not necessary, and at the same time they should be able to describe the system in detail and without ambiguity.

There are many ways to conceptually model a system. For small systems, and where a small number of people is involved in the modeling process, this can be done with in-house methods. All the people involved get together and agree on a modeling technique, which can be graphic or textual, computer-based or hand-written.

As systems grow larger and evolve over time, these techniques become hard to maintain and they cannot describe all the things that needed to be modeled in the system. Furthermore, these models would only be understood by the people that are familiar with the ad-hoc modeling technique, adding more problems when changes are involved or when the model has to be shared with other people.

As the systems become more complex, these problems become ever more acute. System architects and designers, who create models and use them to communicate ideas, consequently realized that there is a need to establish methodologies that can describe their domain of interest, which all designers can use and understand, like an Esperanto of modeling. Although this task seems simple, time has shown that it is extremely complicated. An example of this process can be seen in the field of Software Engineering.

Since the early days of computers, "software researchers and developers have been creating abstractions that help them program in term of their design intent rather than the underlying computing environment... and shield them from the complexities of these environments" [37]. These abstractions began as programming languages used to abstract the hardware where they ran. The 1970's saw the creation of *structured methods* to model the data and functional, behavioral aspects of the systems [5]. One main drawback of these methods was the lack of consistency between the data and the behavior parts of the system, as well as between the concepts expressed by the model and the ones provided by the real world or the implementation. In the 1980s there was

an increased focus on Computer Aided Software Engineering (CASE) tools and methodologies to express software design in graphical representations such as state charts, data flow diagrams and others [17]. Although this field attracted great attention from the research community, it was not widely used in the software development process in industry because it did not scale up to handle complex problems, not did it support concurrent engineering. CASE tools have thus become usable by software designers mainly to document their decisions and guide the developers in their work. With the advance of software languages, the abstraction gap between the programmer and the machine increased. Modeling methodologies were necessary to abstract the software under development.

The most common abstraction was the Object-Oriented paradigm, which stipulates that every entity in the system is an object which owns operations, and the behavior in the system occurs by message passing between objects in the system. This paradigm was further supported by third generation languages such as C++ and Java, which have become the de-facto standard for programming languages. Although this paradigm was a great leap forward, "the success of object-oriented modeling approaches was hindered in the beginning of the 1990's by the fact that surely more than fifty object-oriented modeling approaches claimed to be the right one, the so-called object-oriented method war. This "method war" came to a (temporary) end by an industrial initiative, which pushed the development of the meanwhile standardized object-oriented modeling language UML" [11].

After a decade of UML use, it has not proven to be a panacea. Although UML does give a guideline for designing systems, as it has many drawbacks and problems. There are many other system modeling techniques less popular than UML that are also used to model software systems, such as the Z-notation [42], Petri-nets [1], and plain prose, but none of them is currently a major winner in the modeling arena. As stated in [27]"If we built buildings the way we built software, we would be unable to connect them, change them or even decorate them easily to fit new uses; and worse, they would constantly be falling down... We have very little excuse to build software without first doing careful design work".

Although system modeling languages can help create better products, most current modeling languages are not formal – they lack the mathematical foundation for checking the syntax of the model created using them. While this may be tolerable in the initial design of a system, where the architects and designers interact and

communicate their models to each other via meetings, reviews, etc., once a system has lived for some time, its model drifts away from reflecting the system and tends to become incorrect, because of lack of updates, or erroneous updates due to lack of a formal definition of the modeling language. The formality provides the model users with an exact definition of their model.

The formality requirement is far from being trivial. A model should not just be correct and complete; one of its most important characteristics is that it be comprehensible. Most of the models are incomplete because we want to remove clutter from them. Many modeling languages have therefore opted for user-friendliness as their main goal and did not care for the formality requirement. An ideal modeling language needs to be both comprehensible and formal. These two requirements are potentially in odds with each other, so the challenge is to design the language so that it caters to both these objectives.

## ***1.1 Current Software Modeling and Specification Languages***

In this section we briefly describe main approaches and languages for modeling and specification of systems in general and software systems in particular.

### **1.1.1 Free Prose**

Using the free prose approach, the system or software engineer writes a document, often based on a predefined template, that defines the system, using his/her own language and possibly a predefined terminology that the organization uses. The system engineer can also add diagrams that graphically expand the textual definition. Because of the complexity of the systems, there is usually more than one document that defines it, and the system engineers use references between documents to link their relations and interactions. Although this approach is prone to incompleteness, ambiguities, and inconsistencies, it is widely used.

### **1.1.2 Visual Languages**

A large amount of research has been done on how to model complex systems (and specifically software systems) using many kinds of visual languages, e.g., [21], [17], [18], [38]. While most of these methods are used purely for academic purposes, they have provided many ideas that are later implemented in languages and methodologies like UML and OPM.

### 1.1.3 UML and SysML

Created by the Object Management Group and accepted as an industry standard since 1997, "the Unified Modeling Language (UML) is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains and implementation platforms." [31]. UML provides a collection of 13 different diagram types [32] to define the structure and behavior of the system. Although it is probably the most widely used modeling language, UML has been found to have many drawbacks, mainly its complexity (the UML Infrastructure document is 220 pages long, the Superstructure—732 pages long), imprecise semantics and a problematic learning curve. A few citations illustrate this state of affairs:

"The numerous modeling concepts, poorly defined semantics, and lightweight extension mechanisms that UML provides make learning and applying it in a model-driven development environment difficult" [13].

"UML 2.0 lacks both a reference implementation and a human-readable semantic account to provide an operational semantics" [45].

"The infrastructure of UML is gratuitously complex and difficult to maintain" [22].

"The lack of precise semantics for OO notations can result in situations where a reader's interpretation is not the same (or is not consistent) with the model creator's interpretation" [12].

Efforts to provide UML or a subset of UML with formal semantics include attempts that use graph grammars [24], [25], [15], [49], [14], [23], a mathematical notation [2], Abstract State Machines [20], Petri Nets [43], Z-notation [3], [29], B language [39] and UML itself [44]. A survey of the work done on this field is provided in [28].

A descendant of UML, defined as an extension of a UML subset, is the Systems Modeling Language (SysML) [30] also provided by the Object Management Group.

SysML is an initiative originating in the International Council for System Engineers (INCOSE) to customize UML for system engineering, using a subset of the existing diagrams (seven in all) and adding two more diagrams (requirements and parametric). Although this is a big improvement on the side of simplicity, SysML still "does not solve the question of lack of semantics in UML" [46]

### **1.1.4 Object Process Methodology**

Object Process Methodology [7] is a holistic modeling approach that "maintains the balance between system structure and behavior" [6] by treating objects and processes as primary entities in the model, unlike the Object Oriented (OO) approach, where a process cannot exist detached from an object. To model the behavior of the system, OPM uses links that connect objects and processes. Together, the objects, processes and links coexist in a single model that integrates both the structural and procedural aspects of the system, a trait which "reinforces the user's ability to construct, grasp, and comprehend the system as a whole and at any level of detail" [35].

Because of its intrinsic integration of structure and behavior, OPM provides a solid basis for modeling complex systems, and has been extended to model real-time systems [34], [33], ERP [39] and web applications [36]. OPM is described in detail in section 2.

### **1.1.5 Formal Modeling Languages**

There are several formal languages for system modeling and design, some for general systems and others for specific types of systems. Two prominent formal ones are described briefly below.

- 1) Petri Nets [1]: A formal graphical mathematical representation of distributed systems, which serves to demonstrate many properties of the model, like the ordering of events in a network, concurrency and conflicts.
- 2) The Z-notation [42]: A mathematical notations to describe the data types that exist in a system and predicate logic to describe the operations that can occur and how they affect the data in the system.

The basic problem with these formalities is that they are constrained to specific fields (e.g., Petri nets is suited for distributed computing) or their notation is complex (as in the Z-notation), and are not sufficiently abstract to define any kind of system at its different levels. Moreover, these languages are also fairly complicated, making it hard to "see the forest".

## ***1.2 Research Goal***

In a language, the syntax defines how the language is constructed, what combinations of elements are legal. The semantics of the language give the language its meaning.

For example, the English sentence "Cows eat grass." is both syntactically and semantically correct, the sentence "Cows eat airplanes." is correct syntactically, but not semantically, specifically because cows do not eat airplanes. The sentence "The eats cow grass." is neither semantically nor syntactically correct.

The syntax of OPM carries some of its semantics, because unlike written languages where the symbols by themselves have no meaning, in OPM the symbols carry semantics.

The goal of this research is to devise a formal methodology for defining the syntax of an Object-Process Diagram (OPD) and the semantics associated with this syntax.

Although the OPM modeling language is thoroughly defined in [7]and [35], the definitions are not complete and leave some room for the designer to make mistakes that render the model inconsistent. This work complements the previous works by adding a layer of formality that makes a contribution towards converting OPM from a semi-formal to a formal modeling language.

## **2. Object-Process Methodology**

Object-Process Methodology (OPM) is a holistic modeling approach that combines the structure and behavior of the system in the same model, providing full integration of the important system aspects. OPM has been found [41] to be an ontologically complete modeling language according to the Bunge-Wand-Weber framework [47], a theoretical framework for understanding the modeling of information systems, since it is able to model all things in the real world that are of interest to users of information systems.

OPM is defined by its reflective metamodel [35]. A metamodel is a model of a methodology, which provides further understanding of the modeling language and provides a robust basis for code generation, model transformation and analysis.

### **2.1 Concepts**

The primary elements of OPM are entities and links. Entities are the generalization of things and states, and things are a generalization of object and process – the two primary building blocks in an OPM model. At any time, a stateful object (object with states) is at a specific state, and the state of the object is changed through a process. Likewise, links are a generalization of structural and procedural links. Structural links represent the static relation between pairs of things in the system while procedural links express the dynamics of the system.

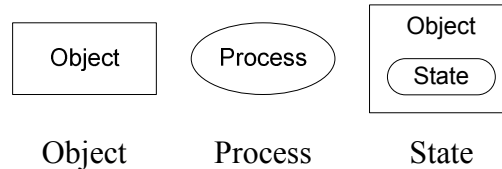
To manage complexity, OPM provides three ways to refine/abstract the system model: in-zooming/out-zooming allows the abstraction of complex entities by hiding its components at high abstraction modeling levels and showing them when their details are required; unfolding/folding provides a means to model any thing in the system as showing or hiding the hierarchy beneath a root thing; and state expressing/suppressing gives freedom to show or hide the states of an object as desired. These mechanisms enable OPM to specify and refine the system indefinitely to any desired level of detail without losing legibility and maintaining simplicity at every detail level.

OPM models a system in two parallel representations, or modalities – one graphic and one textual, which jointly and individually express the same OPM model. A set of Object-Process Diagrams (OPDs) provide a graphical representation of OPM. Each

OPM element in an OPD is denoted by a specific symbol, and in the diagrams the entities are interconnected by entity links following the rules described in the OPM metamodel. The Object-Process Language (OPL) is the textual counterpart of the OPD graphical representation, which describes the model in a subset of English. According to Mayer's cognitive theory [26], the combination of graphical and textual representation increase the processing capabilities of humans and therefore their understanding of the modeled system.

## 2.2 Entities – the OPM Building Blocks

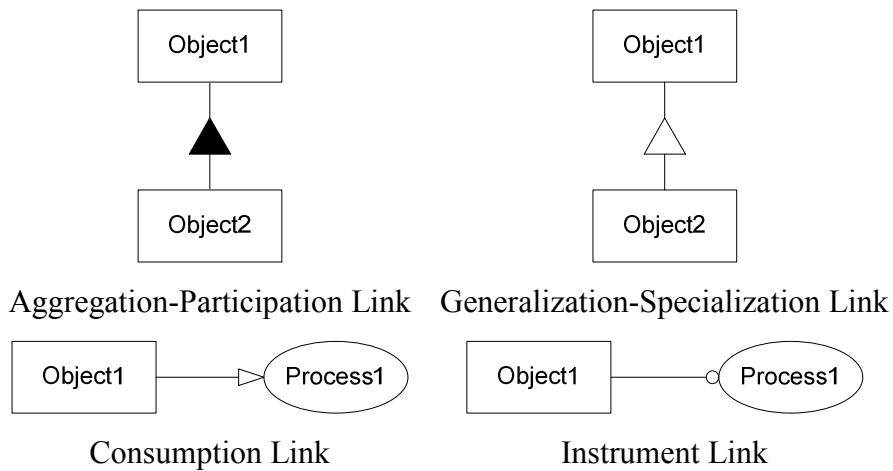
As stated, OPM consists of two types of elements: entities and links. Entities are specialized into things and states, and things can be objects or processes, which are the basic building blocks of OPM. A state is not a stand-alone entity, as it reflects the situation of an object and it is "owned" by it. The status of an object (its state) can be affected only by a process. Objects are represented in OPM as rectangles, and processes as ellipses. A state is represented as a "roundtangle" (rounded edge rectangle) within the rectangle of its owning object, as shown in Figure 1.



**Figure 1. The OPM entities**

In OPM, an object is a thing that exists. A process is a thing that transforms at least one object. Transformation is object generation or consumption or change in the state of the object.

A link is an element that connects two entities and represents a semantic relation between them. Links can be of two kinds: structural and procedural. A structural link represents a static structural relation between two entities, such as aggregation or generalization. A procedural link connects an entity with a process to denote a dynamic behavioral flow of information, material, or others. A further specification of a procedural link is an event link, which indicates a specific event that happens at a particular moment or when specific preconditions are met. Each link is drawn as a line with a special symbol attached to one end or in the middle of the line depending on the link type. Some links types are drawn in Figure 2.



**Figure 2. Example of OPM links**

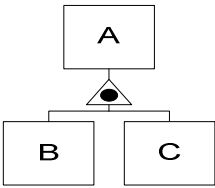
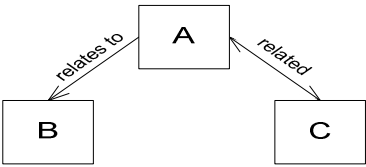
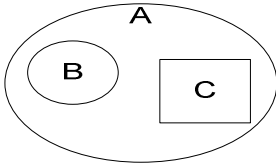
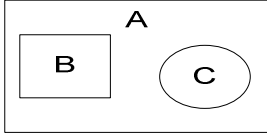
OPM elements have many other attributes, for example essence, which can be either physical – the modeled element is a physical object in the real world, or informatical – something that is not tangible but can be defined and used as a modeling element. Examples of physical elements are *machine*, *raw material* and *product*; Examples of informatical elements are *computing*, *account* and *transaction*.

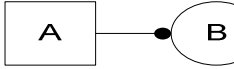
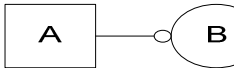
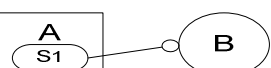
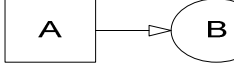

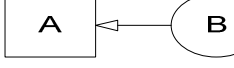
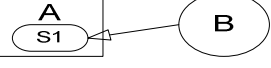
The common constructs of OPM are shown in the following tables.

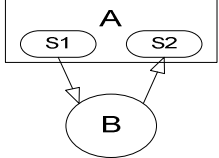
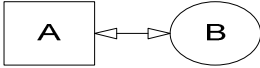
Entities				
Name	Symbol	OPL	Definition	
Things	Object		B is physical. (shaded rectangle)	An object is a thing that exists.  A process is a thing that transforms at least one object.  Transformation is object generation or consumption, or effect—a change in the state of an object.
			C is physical and environmental. (shaded dashed rectangle)	
	Process		E is physical. (shaded ellipse)	
			F is physical and environmental. (shaded dashed ellipse)	

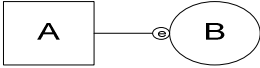
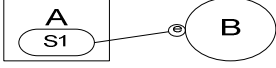
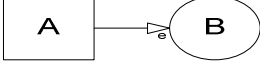
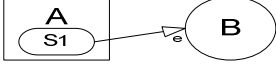
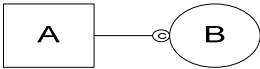
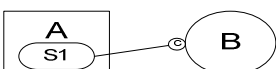
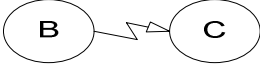
State		<p>A is s1.</p> <p>B can be s1 or s2.</p> <p>C can be s1, s2, or s3.</p> <p>s1 is initial.</p> <p>s3 is final.</p>	<p>A state is situation an object can be at or a value it can assume.</p> <p>States are always within an object.</p> <p>States can be initial or final.</p>
-------	--	--	---

<b>STRUCTURAL LINKS &amp; COMPLEXITY MANAGEMENT</b>				
	<b>Name</b>	<b>Symbol</b>	<b>OPL</b>	<b>Semantics</b>
Fundamental Structural Relations	Aggregation-Participation		A consists of B and C.	A is the whole, B and C are parts.
			A consists of B and C.	
	Exhibition-Characterization		A exhibits B, as well as C.	Object B is an attribute of A and process C is its operation (method).  A can be an object or a process.
			A exhibits B, as well as C.	
	Generalization-Specialization		B is an A. C is an A.	A specializes into B and C.  A, B, and C can be either all objects or all processes.
			B is A. C is A.	

	Classification-Instantiation		B is an instance of A. C is an instance of A.	Object A is the class, for which B and C are instances. Applicable to processes too.
	Unidirectional & bidirectional tagged structural links		A relates to B. (for unidirectional) A and C are related. (for bidirectional)	A user-defined textual tag describes any structural relation between two objects or between two processes.
	In-zooming		A exhibits C. A consists of B. A zooms into B, as well as C.	Zooming into process A, B is its part and C is its attribute.
			A exhibits C. A consists of B. A zooms into B, as well as C.	Zooming into object A, B is its part and C is its operation.

ENABLING AND TRANSFORMING PROCEDURAL LINKS				
	Name	Symbol	OPL	Semantics
Enabling Links	Agent Link		A handles B.	Denotes that the object is a human operator.
	Instrument Link		B requires A.	"Wait until" semantics: Process B cannot happen if object A does not exist.
	State-Specified		B requires s1 A.	"Wait until" semantics: Process B cannot happen if object A is not at state s1.
Transforming links	Consumption Link		B consumes A.	Process B consumes object A.
	State-Specified Consumption Link		B consumes s1 A.	Process B consumes object A when it is at state s1.
	Result Link		B yields A.	Process B creates object A.
	State-Specified Result Link		B yields s1 A.	Process B creates object A at state s1.

Input-Output Link Pair		B changes A from s1 to s2.	Process B changes the state of object A from state s1 to state s2.
Effect Link		B affects A.	Process B changes the state of object A; the details of the effect may be added at a lower level.

EVENT, CONDITION, AND INVOCATION PROCEDURAL LINKS			
Name	Symbol	OPL	Semantics
Instrument Event Link		A triggers B. B triggers A.	Existence or generation of object A will attempt to trigger process B once. Execution will proceed if the triggering failed.
State-Specified Instrument Event Link		A triggers B when it enters s1. B requires s1 A.	Entering state s1 will attempt to trigger the process once. Execution will proceed if the triggering failed.
Consumption Event Link		A triggers B. B consumes A.	Existence or generation of object A will attempt to trigger process B once. If B is triggered, it will consume A. Execution will proceed if the triggering failed.
State-Specified Consumption Event Link		A triggers B when it enters s1. B consumes s1 A.	Entering state s1 will attempt to trigger the process once. If B is triggered, it will consume A. Execution will proceed if the triggering failed.
Condition Link		B occurs if A exists.	Existence of object A is a condition to the execution of B.  If object A does not exist, then process B is skipped and regular system flow continues.
State-Specified Condition Link		B occurs if A is s2.	Existence of object A at state s2 is a condition to the execution of B.  If object A does not exist, then process B is skipped and regular system flow continues.
Invocation Link		B invokes C.	Execution will proceed if the triggering failed (due to failure to fulfill one or more of the conditions in the precondition set).

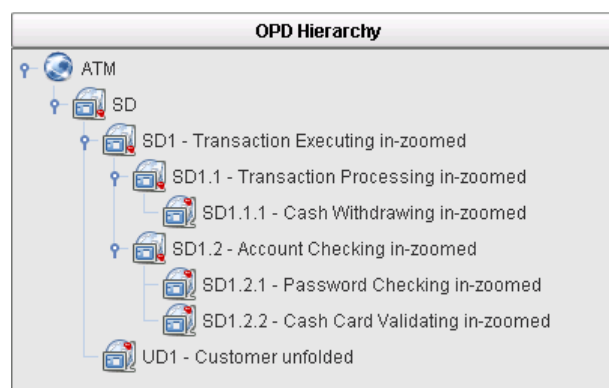
### 2.3 Object Process Diagram

A System Model is an OPM model that defines a system. A system model consists of a set of Object Process Diagrams (OPDs) arranged in a tree structure. The OPDs in a system model are interconnected by the in-zooming or unfolding relation. At any stage in the modeling process, the modeler can decide to increase the details for a specific thing in a model, and this is done using the in-zooming and unfolding operations.

When a thing is in-zoomed a new OPD is created, in which the in-zoomed thing is enlarged and centered in the diagram. When a thing is unfolded, the operation creates a new OPD where the unfolded thing is located at the top of the diagram.

The in-zooming and unfolding operations create an OPD hierarchy. The OPD hierarchy starts with the topmost level system diagram, called SD. Each diagram that is spawned from this diagram using the in-zooming operation is named SD1, SD2... sequentially. Furthermore, the modeler can also in-zoom a thing that exists in one of these SDs (for example SD2), creating a new system model which is called SD2.1. Additional models that are spawned from this model using the in-zooming operations will be named SD2.2, SD2.3 and so on. Generally, the level of an SD is the number of numbers separated by dots that it has in its name (SD has no numbers therefore its level is 0). When an in-zooming operation creates a new SD, this SD is named by adding a dot to the name of the SD from which this one is created and adding the lowest sequence number that is not yet used in the system model.

The unfolding operation also spawns new SDs, but unlike in-zooming, these SDs occur below the topmost level diagram (SD) and are named UD1, UD2... sequentially. An example of an OPD Hierarchy is shown in Figure 3.



**Figure 3. OPD hierarchy example**

From the hierarchy shown above the following things can be observed:

- 1) The ATM System Model consists of 7 system diagrams.
- 2) The topmost level diagram SD contains one in-zoomed thing: *Transaction Executing*.
- 3) SD1 contains two things that are in-zoomed: *Transaction Processing* and *Account Checking*.
- 4) SD1.1 contains one in-zoomed thing: *Cash Withdrawing*.
- 5) SD1.2 contains two in-zoomed things: *Password Checking* and *Cash Card Validating*.
- 6) UD1 is the unfolded diagram of thing Customer

### 3. Graph Grammars

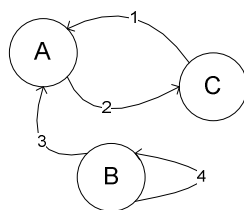
Graph Grammars (or Graph Transformations) is a field of Graph Theory that formalizes the creation or transformation of graphs using predefined transformation rules. Following are definitions of basic Graph and Graph Grammar concepts. Unless otherwise stated, all definitions are taken from [4] and [10].

#### 3.1 Graphs

##### Definition 1 Labeled Graph

- Given two fixed alphabets  $\Omega_V$  and  $\Omega_E$  for node and edge labels respectively, a **labeled graph** is a tuple  $G = \langle G_V, G_E, s^G, t^G, lv^G, le^G \rangle$  where
  - $G_V$  is a set of *vertices* (or *nodes*)
  - $G_E$  is a set of *edges* (or *arcs*)
  - $s^G, t^G : G_E \rightarrow G_V$  are the *source* and *target* functions, and
  - $lv^G : G_V \rightarrow \Omega_V$  and  $le^G : G_E \rightarrow \Omega_E$  are the *node* and the *edge labeling* functions respectively.

The most common way to picture a graph is by drawing a circle for each vertex and joining these circles by a line for every edge that connects between the nodes, as shown in Figure 4.



**Figure 4. Basic graph**

The graph in Figure 4 has 3 nodes and 4 edges. The names given to the nodes and edges in the graph here are taken from their labels. Mathematically the graph consists of a set of nodes  $G_V = \{v_1, v_2, v_3\}$  and a set of edges  $G_E = \{e_1, e_2, e_3, e_4\}$  (the name of the nodes and edges was chosen arbitrarily. They will be later mapped to their label by the labeling functions), functions  $s^G = \{(e_1, g_3), (e_2, g_1), (e_3, g_2), (e_4, g_2)\}$  and  $t^G = \{(e_1, g_1), (e_2, g_3), (e_3, g_1), (e_4, g_2)\}$  for source and target node matching, and

functions  $lv^G = \{(g_1, A), (g_2, B), (g_3, C)\}$  and  $le^G = \{(e_1, 1), (e_2, 2), (e_3, 3), (e_4, 4)\}$  for edge and node labeling, over the alphabets  $\Omega_V = \{A, B, C\}$  and  $\Omega_E = \{1, 2, 3, 4\}$ .

An Object-Process Diagram (OPD) can be considered a Directed Typed Graph as follows:

- $\Omega_V^{OPM} = \{Object, Process, State\}$  – the node alphabet.
- $\Omega_E^{OPM} = \{Object-State, Tagged, Aggregation-Participation, Exhibition-Characterization, Generalization-Specialization, Classification-Instantiation, Agent, Instrument, Consumption, Result, Effect, Input-Output pair, Invocation, Event, Consumption Event, Condition, Exception\}$  – the edge alphabet.

Although this is the formal notation for the OPM graph, its graphical OPD representation is easier to understand and more straightforward, so this is the notation used throughout this work. Furthermore, OPM states are not stand-alone entities and are drawn inside the object that owns them. The object ownership relation is abstracted by adding a new type of link, *Object-State*, which denotes that it belongs to the object. Since the notation in OPM is more expressive and is part of the OPD syntax, we use it, bearing in mind that it can be changed to the formal node-link-node representation by detaching the state from the owning object, moving it out of the object and adding a link between the object and the detached state.

An OPM model is a graph of graphs – a tree of OPDs, where each node in the tree is an OPD, and the links in the tree are defined by the refinement/abstraction (in-zooming/out-zooming or unfolding/folding) relations between the OPDs.

To be able to use graph grammars we define additional properties of graphs and operations on them.

**Definition 2 Graph Morphism**

- Let  $G = \langle G_V, G_E, s^G, t^G, lv^G, le^G \rangle$  and  $G' = \langle G'_V, G'_E, s^{G'}, t^{G'}, lv^{G'}, le^{G'} \rangle$  be two graphs over the same label alphabets  $\Omega_V$  and  $\Omega_E$ . A Graph Morphism  $f : G \rightarrow G'$  is the pair of functions  $f = \langle f_v : G_V \rightarrow G'_V, f_e : G_E \rightarrow G'_E \rangle$  that preserve sources, targets and labels, such that:
  - $\forall e \in G_E, f_v(s^G(e)) = s^{G'}(f_e(e))$  (source node preservation)
  - $\forall e \in G_E, f_v(t^G(e)) = t^{G'}(f_e(e))$  (target node preservation)
  - $\forall v \in G_V, lv^G(v) = lv^{G'}(f_v(v))$  (node label preservation)

- $\forall e \in G_E, le^G(e) = le^{G'}(f_e(e))$  (edge label preservation)

**Definition 3 Subgraph**

- Suppose  $V, E$  and  $E'$  are sets,  $E' \subseteq E$  and  $s$  is a mapping  $s : E \rightarrow V$ . The operator  $s' = s|_{E'}$  defines a new mapping  $s'$  for every element in  $E'$ , where  $s'(e \in E') = s(e \in E)$ . The value of  $s'(e \in E \setminus E')$  is not defined in this mapping.
- Let  $G$  be a labeled graph as defined in Definition 1.
- A **subgraph**  $S = \langle S_V, S_E, s^S, t^S, lv^S, le^S \rangle$  of  $G$ , written as  $S \subseteq G$ , is a graph having  $S_V \subseteq G_V, S_E \subseteq G_E, s^S = s^G|_{S_E}, t^S = t^G|_{S_E}, lv^S = lv^G|_{S_V}$  and  $le^S = le^G|_{S_E}$ .

**Definition 4 Partial Graph Morphism**

- A partial graph morphism  $m : G \rightarrow G'$  is a graph morphism of a subgraph of  $G$  to a subgraph of  $G'$ .

### 3.2 Basic Graph Grammar Concepts

Based on the above definitions, we now formally define a *Graph Transformation Rule*. There are two common ways to define transformation rules: the double pushout approach (DPO) and the single pushout approach (SPO). The double pushout approach has stronger properties than the single pushout approach but it is more complex. Since the single pushout approach satisfies all our requirements, we use it for simplicity.

**Definition 5 SPO Production Rule**

- A production  $p : L \xrightarrow{r} R$  consists of a production name  $p$  and an injective partial graph morphism  $r$  called the production morphism.  $L$  and  $R$  are graphs called the left-hand graph and the right-hand graph, respectively.

Informally, the left-hand graph describes the context needed for the rule to be applied – nodes and edges alike. The right-hand graph shows how the original part of the graph will look like after the application of the production rule. The morphism specifies which element (node or edge) in the left-hand graph is matched with which element in the right-hand graph. Each element missing from the right-hand graph is deleted from the graph in which the production is applied, and if this deletion causes dangling edges, i.e., edges that have no source or destination, they are deleted as well. Elements missing from the left-hand graph that exist in the right-hand graph are added

to the graph where the production is applied. This operation is called a *Derivation*, as defined below.

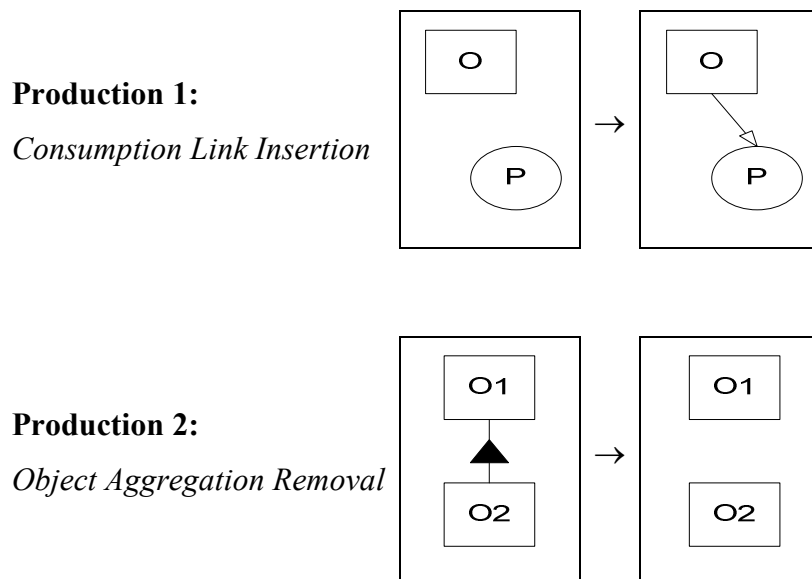
**Definition 6** *Production, Derivation*

- A match for  $p : L \xrightarrow{r} R$  in some graph  $G$  is a graph morphism  $m : L \rightarrow G$ . Given a production  $p$  and a match  $m$  for  $p$  in graph  $G$ , the **direct derivation** from  $G$  with  $p$  at  $m$ , written  $G \xRightarrow{p,m} H$ , is done as follows:
  - Using morphism  $m$ , delete vertices and edges of  $G$  that occur in  $L$  and do not occur in  $R$ .
  - Add to  $G$  all vertices and edges that occur in  $R$  but do not occur in  $L$ .
  - Delete all dangling edges from  $G$ .

Intuitively, the application of a graph production  $p : L \xrightarrow{r} R$  to a graph  $G$  works as follows: Replace the occurrence (match) of  $L$  in  $G$  by  $R$ . Delete edges whose source or target nodes are deleted. If a node or an edge is supposed to be deleted as well as preserved, solve this conflict by deletion too.

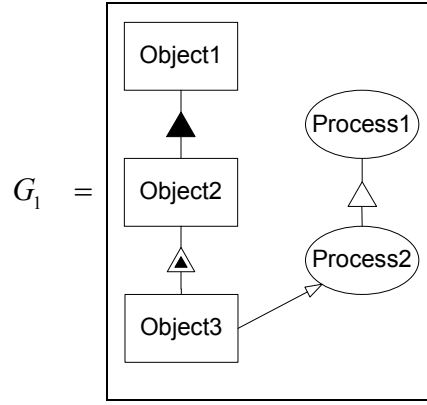
**Example 1**

The following example demonstrates the basic graph grammar concepts on a small OPM model. Throughout this work, each OPD (such as the left-hand and right-hand graphs) is surrounded by a rectangular frame. The grammar's set of production rules is given in Figure 5.



**Figure 5. The two productions of Example 1**

The initial graph for our example is  $G_1$  shown in Figure 6.



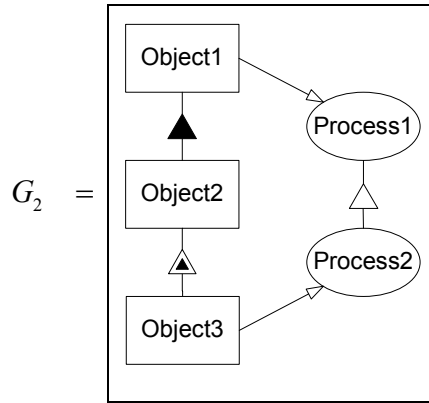
**Figure 6. Initial graph of Example 1**

Formally,  $G_1 = \langle G_V, G_E, s^G, t^G, lv^G, le^G \rangle$  over  $\Omega_V^{OPM}$  and  $\Omega_E^{OPM}$ .

- $G_V = \{\mathbf{Object1}, \mathbf{Object2}, \mathbf{Object3}, \mathbf{Process1}, \mathbf{Process2}\}$
- $G_E = \{Ag1, Ex1, Gen1, Cons1\}$
- $s^G = \{(Ag1, \mathbf{Object1}), (Ex1, \mathbf{Object2}), (Gen1, \mathbf{Process1}), (Cons1, \mathbf{Object3})\}$
- $t^G = \{(Ag1, \mathbf{Object2}), (Ex1, \mathbf{Object3}), (Gen1, \mathbf{Process2}), (Cons1, \mathbf{Process2})\}$
- $lv^G = \{(\mathbf{Object1}, Object), (\mathbf{Object2}, Object), (\mathbf{Object3}, Object), (\mathbf{Process1}, Process), (\mathbf{Process2}, Process)\}$
- $le^G = \{(Ag1, Aggregation-Participation), (Ex1, Exhibition-Characterization), (Gen1, Generalization-Specialization), (Cons1, Consumption)\}$

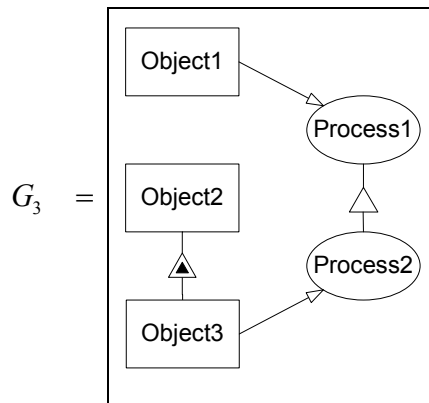
Let us apply the two productions on  $G_1$ , starting with Production 1 – *Consumption Link Insertion*. The first step is to find a subgraph of the graph  $G_1$  that matches the left-hand side of Production 1.

There are six possible matches:  $\{\mathbf{Object1}, \mathbf{Process1}\}$ ,  $\{\mathbf{Object1}, \mathbf{Process2}\}$ ,  $\{\mathbf{Object2}, \mathbf{Process1}\}$ ,  $\{\mathbf{Object2}, \mathbf{Process2}\}$ ,  $\{\mathbf{Object3}, \mathbf{Process1}\}$  and  $\{\mathbf{Object3}, \mathbf{Process2}\}$ . The last option exists since we did not yet add negative constraints to prevent redundant links. We choose to apply the rule on  $\{\mathbf{Object1}, \mathbf{Process1}\}$ , resulting in graph  $G_2$ , shown in Figure 7.



**Figure 7. Example 1 graph after application of Production 1 – Consumption Link Insertion**

Next, we apply Production 2 – *Object Aggregation Removal* on the resulting graph  $G_2$ . The only available match for the left-hand graph is {**Object1**, **Object2**}. Applying this rule, the edge in the left-hand graph is removed so it is missing in the right-hand graph. Therefore we remove, obtaining the graph  $G_3$ , shown in Figure 8.



**Figure 8. Example 1 graph after application of Consumption Link Insertion and Object Aggregation Removal**

### 3.3 Application Conditions

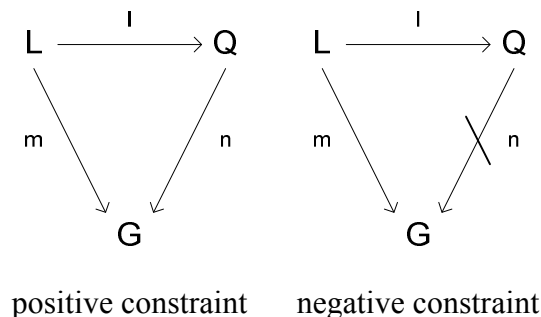
Using derivations, we can describe how graphs are legally transformed into other graphs. However, specifying when these transformations can be applied is limited to the positive application condition, namely, that the graph to be transformed contains a subgraph that matches the left-hand side graph. This positive application condition can be extended with *application conditions* which specify contexts, where the transformation can be applied. Application conditions can be positive (contexts that exist in the graph) or negative (contexts that must not exist in the graph).

To specify application conditions, we specify not just one left-hand side graph  $L$ , but a set of graph morphisms  $\{L \xrightarrow{l} \hat{L}\}$  called *constraints* [19], [9], [16]. Each constraint represents a structure on the left-hand graph that must exist for positive constraints and must not exist for negative constraints. To produce a match that satisfies the constraints, we must first check that all the constraints hold in the source graph, and if so, find a subgraph that matches  $L$ , as defined formally below.

**Definition 7** Application Conditions

- An application condition over a graph  $L$  is a finite set  $A = \{L \xrightarrow{l} \hat{L}\}$  of graph morphisms of the form  $L \xrightarrow{l} \hat{L}$  called constraints.
- Let  $p: L \xrightarrow{r} R$  be a production,  $a: L \xrightarrow{l} Q$  a positive constraint, and  $m: L \rightarrow G$  a match for  $L$  in graph  $G$ . We say that  $m$  satisfies  $a$ , denoted by  $m \models a$  if there exists a graph morphism  $n$  such that  $n \circ l = m$ , where  $\circ$  is the mathematical function composition operator, which means that we apply morphism  $l$  to  $L$  and then we apply morphism  $n$  to the result.
- A negative constraint is defined as a regular constraint for which morphism  $n$  must not exist.
- A match  $m$  satisfies an application condition  $A$  over  $L$ , denoted by  $m \models A$  if it satisfies all the constraints  $a \in A$ .

The concepts in this definition are shown graphically in Figure 9. The match  $m$  for  $L$  is shown as an arrow from  $L$  to  $G$ . On the left-hand of Figure 9 a positive constraint, given by morphism  $l$ , transforms graph  $L$  to  $Q$ . The positive constraint requires the existence of morphism  $n$  (not specified), which matches  $Q$  to  $G$ . The right-hand side of Figure 9 shows the same constellation, except here  $n$  must not exist.



**Figure 9.** Application Conditions

Usually the specified constraints are negative, since the positive constraints can be modeled in the left-hand side graph of the production.

**Definition 8** Conditional Production

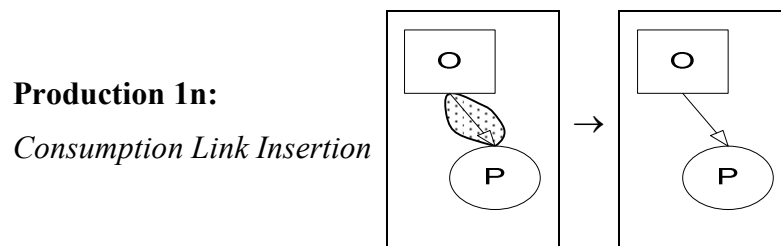
- A conditional production  $\hat{p} = (L \xrightarrow{p} R, A)$  is a pair consisting of a graph morphism  $p$  and an application condition  $A$  over  $L$ .

**Definition 9** Direct Conditional Derivation

- Production  $p$  is applicable to graph  $G$  at  $L \xrightarrow{m} G$  if  $m \models A$ . When this is the case, the direct derivation  $G \xRightarrow{p,m} H$  is called a direct conditional derivation  $G \xRightarrow{\hat{p},m} H$ .

**Example 2**

The above definitions are best understood with an example. Continuing with the transformations in Example 1, we revise Production 1 to include a negative constraint, which removes the possibility of creating duplicate consumption links between two entities. The result is shown in Figure 10.



**Figure 10. Production 1 with a negative constraint**

The rephrased Production 1, called Production 1n, simply means that a consumption link can only be added if a consumption link between the two entities does not yet exist. The addition of the negative constraint does not change the resulting graph in Figure 7, but it rules out the option (Object3, Process2) from the set of possible matches.

## 4. OPD Graph Grammar

There are two possible approaches to maintaining the syntactic legality of an OPD:

- 1) Proactive Verification: Maintaining syntactic legality of the OPD at modeling time by proactively verifying that each modeling operation is legal as it is being executed by the system designer so the model is syntactically legal at any time. This approach is fairly complicated and rather than being helpful, it often encumbers the user, because there may be intermediate situations in which the model needs to be temporarily inconsistent.
- 2) Retroactive Verification: Verifying retroactively, by user request, that the OPD has remained syntactically legal after applying one or more modeling operations to the OPD.

Our syntactic legality algorithm combines the proactive and retroactive verification approaches. We limit the OPD construction process by defining transformation rules stipulating what is and what is not permitted in OPD construction while allowing for temporary inconsistencies and checking them periodically, as requested by the user.

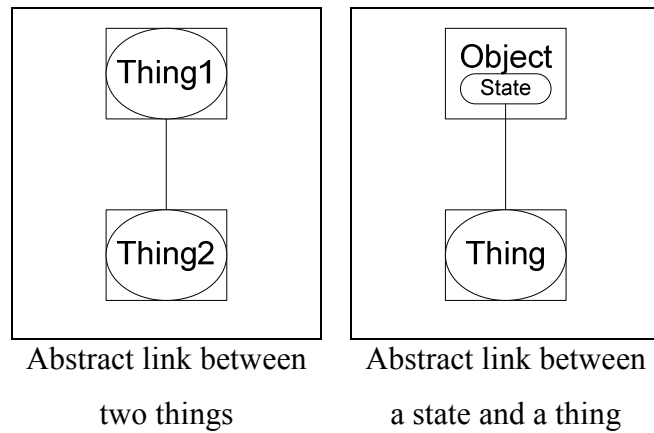
This section describes a graph grammar for creating a system model in one OPD. The zooming and folding OPM capabilities are not handled since for syntactic purposes an OPM model that consists of many OPDs can be recursively converted into a single OPD by "flattening" the OPD hierarchy via successive model element assignments without loss of model information.

### 4.1 Preliminary Definitions

This section introduces graphical representations that abstract OPD elements and are not available in OPM or in the OPM metamodel [35]. These representations are later used in the definitions of the OPM graph grammar productions.

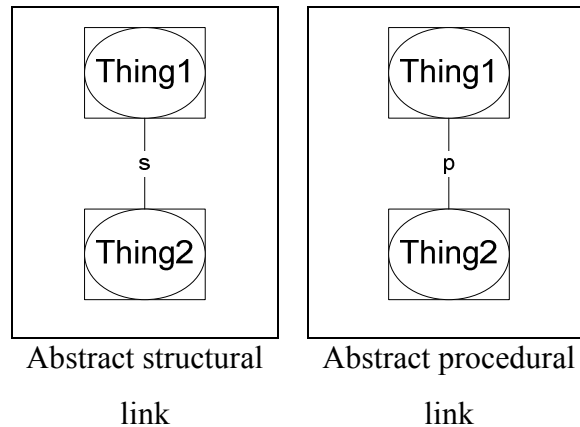
#### 4.1.1 Abstract Link

An abstract link is an OPM link that stands for any type of concrete link that can connect two entities in the model. Its graphical representation is a straight line drawn between the two entities, as shown in Figure 11.



**Figure 11. Abstract links**

An abstract link specializes into an abstract structural link and an abstract procedural link, respectively denoted with "s" and "p" along the abstract link line, as shown in Figure 12.



**Figure 12. Abstract structural and procedural links**

An abstract link is undirected—it has no specified direction, but a direction can be added using an open arrowhead at one of its ends. Since in OPM this is the symbol for the tagged structural relation, the tagged structural relations symbol is changed in the relevant rule to a double arrowhead to remove the ambiguity.

## 4.2 Modeling Conventions

A negative constraint is drawn as shaded areas in the appropriate context within the left-hand graph of the production.

Entities in the rules will be symbolized as follows:

- Thing: T (if only one appearance exists in the OPD), T1, T2 ...
- Object: O (only one appearance), O1, O2 ...

- Process: P (only one appearance), P1, P2 ...
- States: s (only one appearance), s1, s2 ...

This naming convention is used to properly identify the entities in a production.

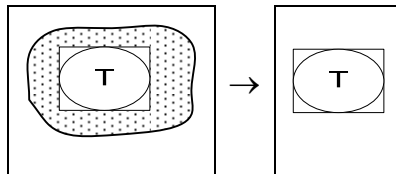
Regarding duplicate thing names, we follow the OPM thing naming convention which states that in any OPM system model there may be exactly one top-level object or process class with any given name. Duplicate names are allowed if and only if the thing is a refineable (part, feature, specialization, or instance) of another thing. We also disallow two copies of the same Thing in the same OPD.

### 4.3 OPD Creation Productions

#### Production 1 Thing Creation

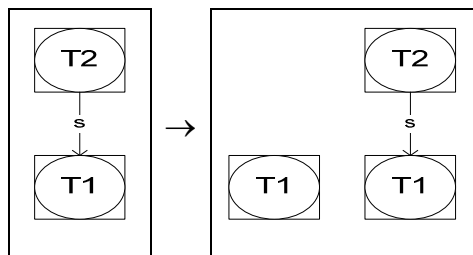
There are two possible productions for the creation of a new thing in an OPD: either there is no thing in the OPD with the same name, or there exists a thing in the OPD with the same name but it is a refineable of an existing thing in the OPD.

The first case is fairly simple, and is shown in Figure 13.



**Figure 13. Thing Creation production**

The second case requires the existence of another thing which is the structural parent of the thing that has the same name as the thing being added. This is shown in Figure 14.

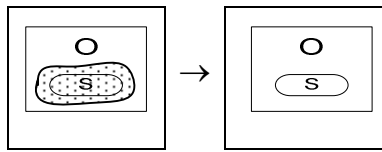


**Figure 14. Existing Name Thing Creation Production**

Because of the limited graphical expressiveness of graph grammars, this second production by itself can only be used if the first production did not provide a match. The correct production (not displayed graphically) states that if there is a thing (no match for left-hand of first production), but this thing is a structural child of another thing (match for left-hand of second production) then the thing can be created.

### Production 2 State Creation

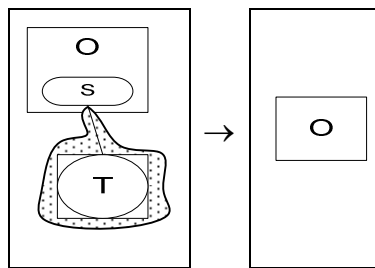
This production, shown in Figure 15, is applicable only to objects. The only constraint here is that the state is not yet a state of the object.



**Figure 15. State Creation production**

### Production 3 State Removal

A state can be removed when there are no links connected to it. The production for this rule is shown in Figure 16.

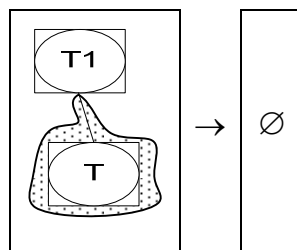


**Figure 16. State Removal production**

This production makes use of the abstract link, which can be of any possible type allowed by the OPD syntax.

### Production 4 Thing Removal

This production is applied to remove a thing from an OPD. As Figure 17 shows, the thing to be removed must not have any link connected to it.



**Figure 17. Thing Removal production**

### Production 5 Link Creation

On top of the semantic differences between the two link types, structural and procedural, they also differ syntactically and therefore are handled separately. Furthermore, there are nuances within each link type that must be taken into account. Production 5 is therefore divided into sub-productions to account for these

differences. Note that when the end of a link is connected to an object (either as source or destination), that end can also be connected to a state within that object, provided the OPD syntax allows for this.

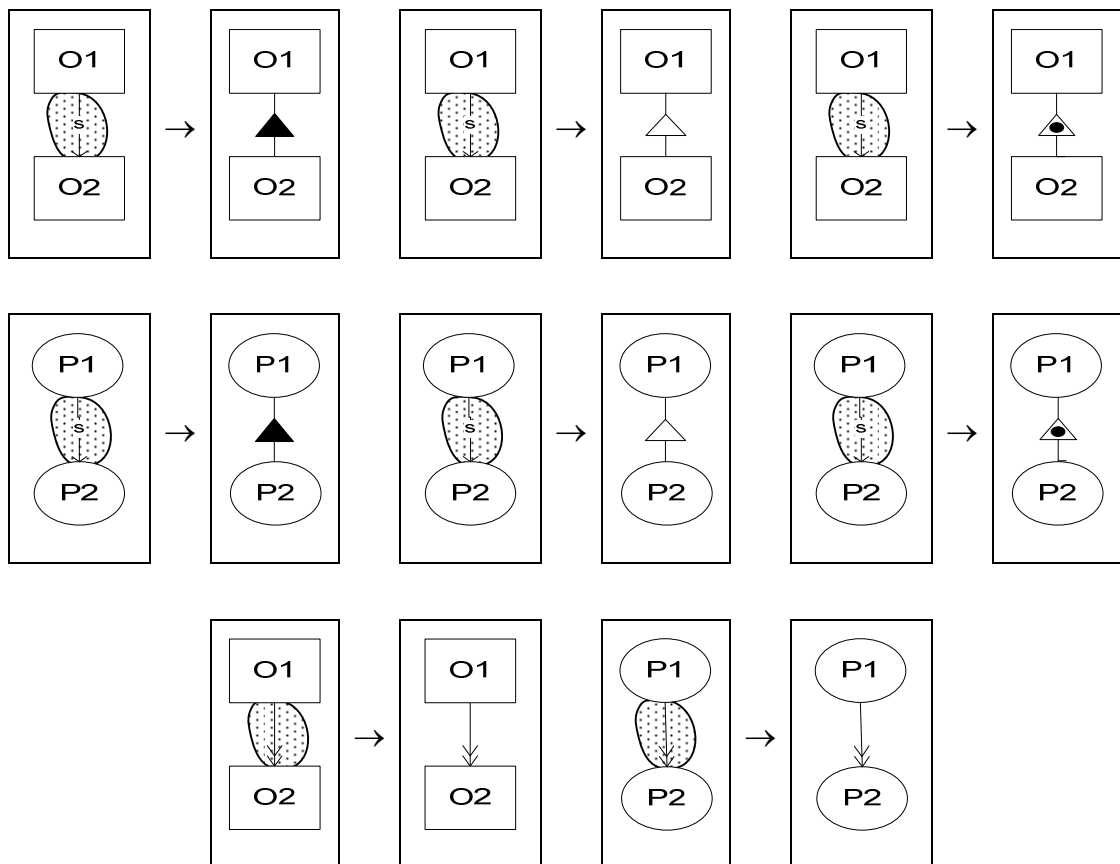
**Production 5.1** Structural Link Creation

Structural links connect things to express static relations. Along the Homogeneity attribute of structural links, they are divided into two basic types: homogeneous and non-homogeneous. Homogeneous links may connect only things of the same persistence (i.e., two objects or two processes), while non-homogeneous links do not have this restriction.

There is one case where there may be two homogeneous links between two things in an OPD. This can occur when an object is both an aggregate and a generalization of another object. This exception will be modeled in a specific rule.

**Production 5.1.1** Homogeneous Structural Link Creation

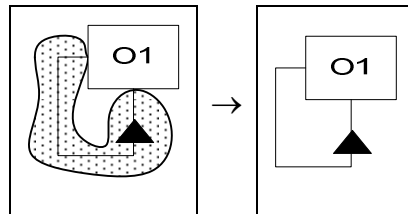
There are three types of homogeneous structural links: Aggregation-Participation, Generalization-Specialization and Classification-Instantiation. Their productions are shown in Figure 18.



**Figure 18. Homogeneous structural link creation productions**

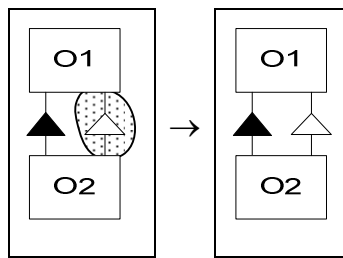
The only constraint for structural links (not only homogeneous) is that only one structural link can connect two Things.

There are two special cases of structural links that do not conform to the previous rules. One special case is the loop link, where a thing is connected to itself. This only occurs for objects and aggregation-participation relations. This production is shown in Figure 19.



**Figure 19. Aggregation loop creation production**

The second special case is the generalization-specialization link, which can link two objects in addition to an aggregation-participation link. The production used to create this relation is shown in Figure 20.

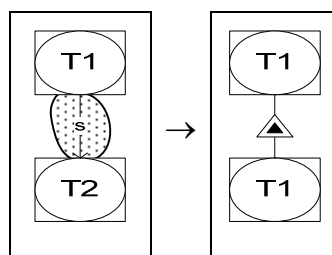


**Figure 20. Generalization and Aggregation link pair creation production**

Note that this rule checks that two generalization-specialization links are not created in parallel between two things.

**Production 5.1.2 Non-Homogeneous Structural Link Creation**

Exhibition-Characterization is the only OPM link that is non-homogeneous, so it can connect any two things. Therefore its production rule is fairly generic. It is shown in Figure 21.



**Figure 21. Non-homogeneous structural link creation production**

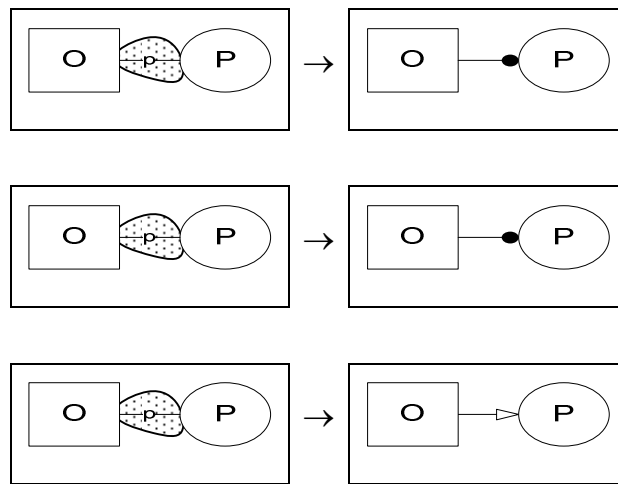
**Production 5.2 Procedural Link Creation**

Procedural links are the elements in OPM that enable the indication of something dynamic that happens in the system. Except for one procedural link (the invocation link) all the procedural links connect an object with a process. The primary constraint for creating a procedural link is that only one procedural link can connect two things in an OPD.

The productions for creating procedural links are divided into four groups: Object-to-Process, Process-to-Object, Bi-Directional, and Invocation.

**Production 5.2.1** Object-to-Process Link Creation

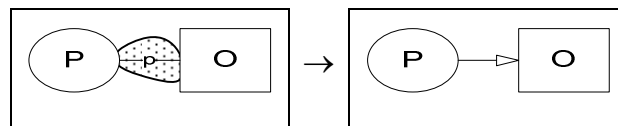
Object-to-Process links include the agent, instrument and consumption links. The source of each one of these links is an object and their destination is a process. The productions for creating these links are shown in Figure 22.



**Figure 22. Object-to-Process link creation productions**

**Production 5.2.2** Process-to-Object Link Creation

Process-to-Object links are links whose source is a process and target is an object. The result link is the only OPM link of this type. The production rule to add this link is shown in Figure 23.

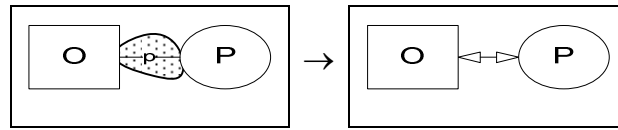


**Figure 23. Process-to-Object link creation production**

**Production 5.2.3** Bidirectional Procedural Link Creation

Although our definition of the OPM graph does not allow for bi-directional links (i.e., every link must have a source and a target), there is one procedural OPM link that is bi-directional – the effect link. Since the direction of the link is irrelevant, it can be

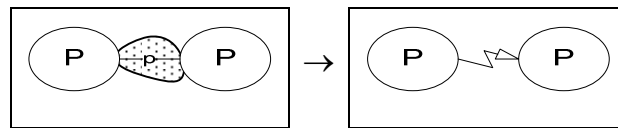
modeled as a link whose source is the object and the target is the process. The production for this link is shown in Figure 24.



**Figure 24. Bidirectional procedural link creation production**

**Production 5.2.4 Invocation Link Creation**

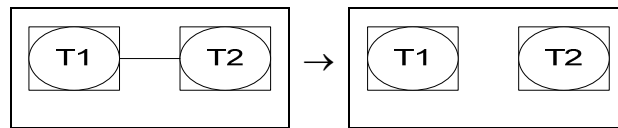
The invocation link connects two processes. The production used to create this link is shown in Figure 25.



**Figure 25. Invocation link creation production**

**Production 6 Link Removal**

A link between two things can be removed at any time, no questions asked, as specified by the production shown in Figure 26.



**Figure 26. Link removal production**

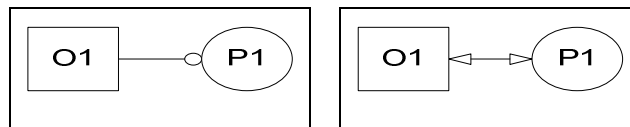
Note the use of the generic link notation, which matches any kind of link that may connect T1 to T2.

## 5. OPD Abstraction

The purpose of OPD abstraction is to validate the model shown in a single OPD. The validation of the model is done by searching for illegal constructs during the abstraction process. If the algorithm does not find any illegal constructs, the OPD is valid in OPM syntax and semantics.

This work is applicable to a single OPD. Note that a single OPD may contain a full model or only a part of a greater system model with other interconnected OPDs.

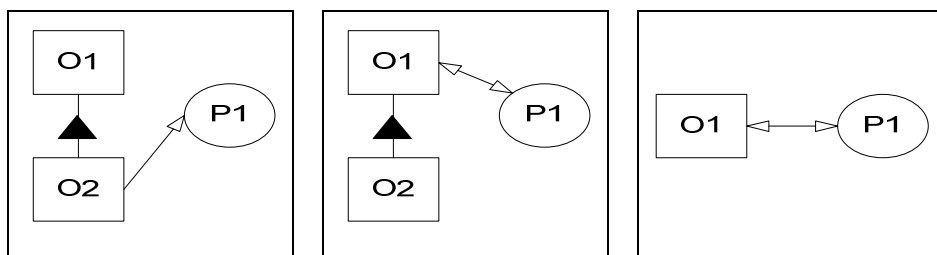
The word abstraction is used in this context differently than in OPM, therefore it needs to be clarified. The word *abstraction* is used to denote reduction of the amount information, while trying to minimize the semantic information that is lost in the process. In the context of an OPD, an element of the OPD is (or can be) abstracted to (or by) another element in the OPD if the new element does not contradict the meaning of the original element, and in addition, there is no other element that can change it without losing more semantic information. The best way to understand this is by an example, shown in Figure 27.



**Figure 27. Abstraction of a simple OPM link**

The left hand side diagram of Figure 27 shows that O1 is an instrument of P1. The semantics of this is that O1 is required for P1's execution. The right hand side diagram changes the instrument link to an effect link. Semantically, an effect link also denoted that P1 requires O1, but it adds more information, by defining that it also changes O1. Therefore effect link can abstract instrument link.

A more complex and realistic example is shown in Figure 28.



**Figure 28. Abstraction of the link of a part**

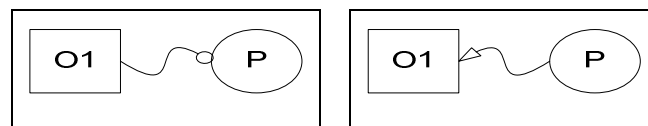
The left hand side diagram of Figure 28 shows that O1 consists of O2, and O2 is consumed by P1. The semantics in this case is trivial. The middle diagram shows an intermediate step in the abstraction process, since the amount of information in the diagram has not been greatly reduced. Semantically, this diagram shows that O1 is changed by P1. Since O1 consists of O2, the meaning of the original diagram is not contradicted, but information on the specific change done to O1 has been lost. The right hand diagram finally reduces the amount of information by deleting O2 from the diagram. The new diagram does not contradict the original diagram and it also contains the semantic information that was initially shown, while some of the information has been lost. This is the meaning of abstraction.

The abstraction of the OPM Model is done by successively applying graph grammar productions to the OPD until no further abstraction can be done (no production rule can be applied). We call the resulting model the **final abstraction** of the OPD. Even though the final abstraction of the OPD is valid as an OPM model, it must be checked by the modeler because only the modeler can validate that the semantics of this model matches the semantics she or he wanted it to represent.

## 5.1 Notation

### 5.1.1 Temporary Link

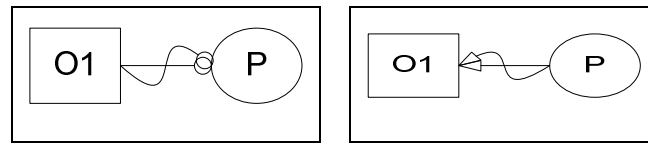
Throughout the execution of the OPD abstraction algorithm, new *temporary links* between existing things in the OPD may be added. Temporary links are links that do not exist in the original OPD; they are added by the OPD abstraction algorithm to denote links whose type can be changed by the algorithm, unlike links that exist in the original OPD and cannot be changed. The semantics of a temporary link is the same as that of its regular counterpart. Graphically, it is denoted with the same edge symbol as its regular link counterpart and a curved (rather than straight) line, as in Figure 29 which shows two temporary links: an instrument link and a result link.



**Figure 29. Temporary instrument link (left) and result link (right)**

In some productions the link can be either temporary or regular, i.e., one that existed in the original OPD or exists as a result of an abstraction round. In these cases the link

is shown as a superposition of the two notations, as shown in Figure 30. This is a shorthand notation used instead of two productions, one for a temporary link and one for an original link.



**Figure 30. Superpositioned temporary and regular links together**

### 5.1.2 Modeling Height

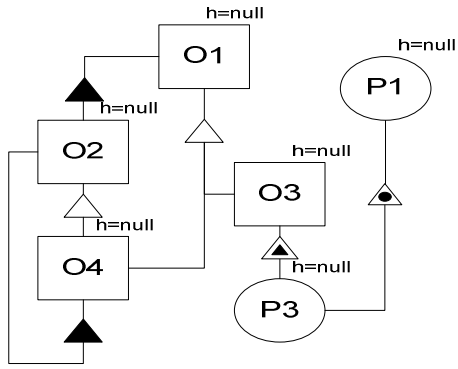
*Modeling height* of a thing denotes how far in the modeling chain is the thing in the OPD. Formally, the modeling height (or height for short) of a thing is greatest Hamiltonian distance (as defined in [7]) of this thing from all the things that have no structural parent. The term *structural parent* of a thing, used in the algorithm, denotes another thing that is the source of a structural link that ends at the first thing. The height of a thing that has no structural parent in is defined to be 0.

The following simple algorithm calculates the height of all the things in an OPD.

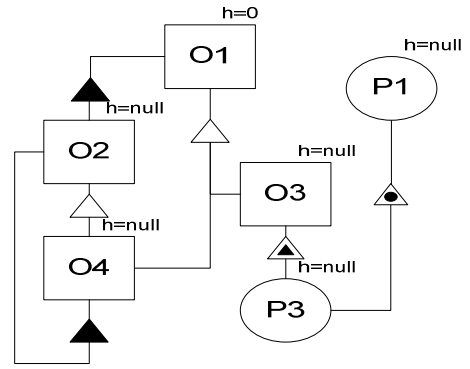
**Algorithm 1** Thing Height Calculation

- Set  $height(\mathbf{thing}) = \text{null}$  for all **things** in the OPD.
- While exists **thing** where  $height(\mathbf{thing}) = \text{null}$ 
  - Select **thing** with no structural parents. If there is no such thing, select **thing** for which all structural parents have  $height(\mathbf{thing}) \neq \text{null}$ 
    - If thing has no structural parents
      - Set  $height(\mathbf{thing}) = 0$
    - Else
      - Set  $height(\mathbf{thing}) = \max(height(\{\mathbf{T}\})) + 1$  where **T** is the set of all structural parents of **thing**.

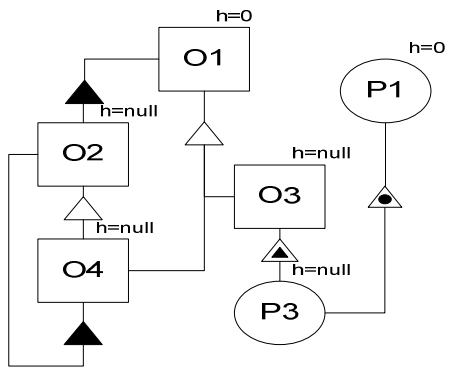
Figure 31 shows an example of the height calculation in a sample OPD.



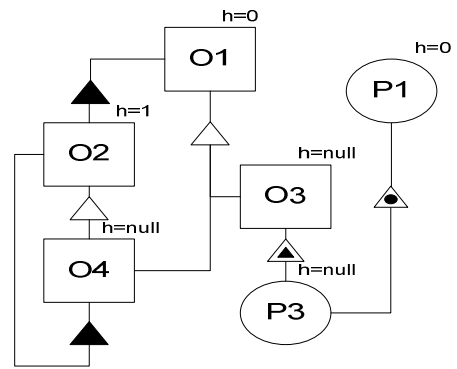
Initial Graph



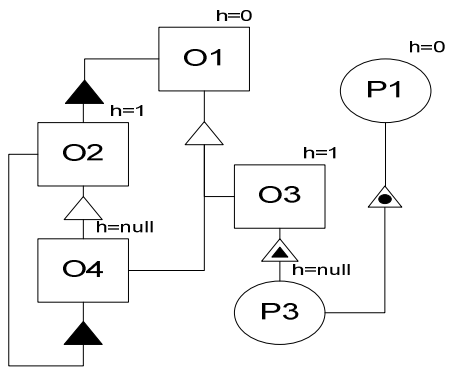
Iteration 1



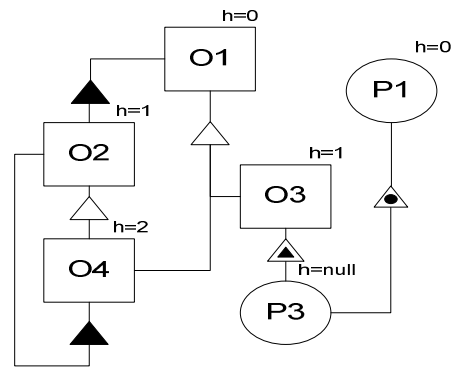
Iteration 2



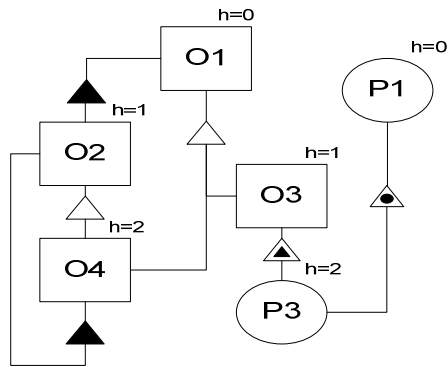
Iteration 3



Iteration 4



Iteration 5



**Figure 31. Calculating the things' height**

It is clear that from the definition of the algorithm, the height of the child of a thing is always greater than the height of its parent. Furthermore, the algorithm calculates the height for all of the things in the diagram, and will always finish since it calculates the height only once for each thing.

**5.2 Aggregation and Exhibition Abstraction**

The OPD abstraction algorithm is based on specific OPD graph grammar productions, which are divided into four groups: *State Change Abstraction*, *State-Specified Link Abstraction*, *Procedural Abstraction*, and *Thing Removal*. These productions are specified below at the end of each one of the four algorithm steps. At the end of each algorithm round the algorithm checks for *Illegal Constructs*. These states are also described after the description of the algorithm. Formally, the OPD Abstraction Algorithm is as follows:

**Algorithm 2 OPD Abstraction Algorithm**

- Input: OPD; Output: abstracted OPD
1. While OPD contains unmarked things:
    - 1.1. Select **thing** having  $\max(\text{height}(\text{thing}))$  from all unmarked things in the OPD and have no outgoing structural links.
    - 1.2. Convert each *temporary link* that starts at **thing** to a *regular link*.
    - 1.3. Apply to **thing** the *State Change Abstraction* production if applicable, as many times as possible.
    - 1.4. Apply to **thing** *State-Specified Link Abstraction* production if applicable, as many times as possible.
    - 1.5. Apply to **thing** *Procedural Abstraction* production if applicable, as many times as possible.
    - 1.6. Check **thing** for *Illegal Constructs*. If *Illegal Constructs* exist, break and return failure on **thing**.
    - 1.7. Apply to **thing** the *Thing Removal* production if applicable. If the production is not applicable, mark **thing**.
  2. Transform all *temporary links* in the OPD to *regular links*.

3. End.

### 5.3 Graph Grammar Productions for Aggregation and Exhibition

#### 5.3.1 State Change and State-Specified Link Abstraction

Both of the following productions are used to transfer the procedural links that start at a state to start at the object that owns the state.

##### Production 7 State Change Abstraction

The first production applied in the OPD Abstraction Algorithm abstracts state changes of an object. This is done as the first step of the algorithm because otherwise the link remaining from an input-output pair may be interpreted as consumption or a result link. The production is shown in Figure 32.

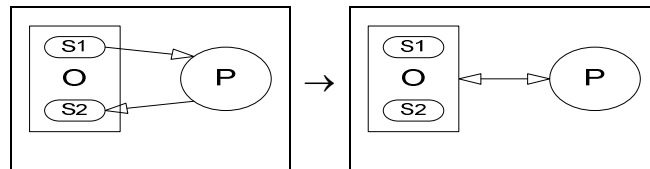


Figure 32. State change abstraction production

##### Production 8 State-Specified Link Abstraction

A state-specified link is a procedural link that starts or ends at a state within an object and is not part of an input-output pair. A state-specified link can be abstracted by migrating its end that touches the state such that it will touch the object that owns (contains) the state. Then the state can be removed.

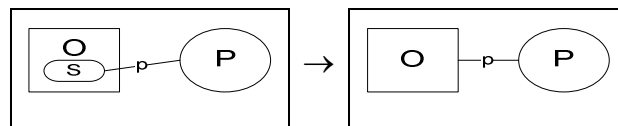


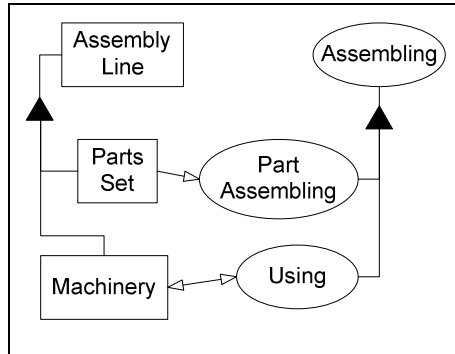
Figure 33. State-Specified Link Abstraction production

Note the use of an abstract procedural link in the production, which means that it can apply to any procedural link.

#### 5.3.2 Procedural Abstraction

The following productions abstract procedural links while maintaining at least an approximation of the original model intent prior to applying structural productions. While this work is syntax-oriented, the productions are determined such that during abstraction, the semantics expressed in the OPD is preserved or abstracted.

The goal of these productions is to abstract procedural links in the structural construct of the OPD. For example, suppose object **Assembly Line** consists of objects **Parts Set** and **Machinery**, and process **Assembling** consists of processes **Part Assembling** and **Using**, having the two procedural links as shown in Figure 34.

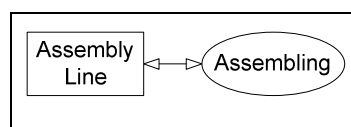


**Figure 34. Assembly Line OPD before abstraction**

**Parts Set** is consumed by **Part Assembling** and **Machinery** is affected by **Using**. The abstraction of this OPD should show only **Assembly Line** and **Assembling**, but if the rest of the diagram is simply deleted, the final result does not reflect the procedural relations between the respective parts of **Assembly Line** and **Assembling**. Therefore, the relations between **Parts Set** and **Part Assembly** and between **Machinery** and **Using** must be "upgraded", or transferred upwards to their respective aggregates **Assembly Line** and **Assembling**.

Starting with **Machinery**, a change in a part object—an object which is a part of another aggregate object (which is the whole) implies change in its aggregate object—the whole. In other words, if **Using** affects **Machinery** then **Using** affects **Assembly Line**. Furthermore, since **Using** is part of **Assembling** then **Assembling** affects **Assembly Line**.

Next, **Parts** is abstracted in the same fashion. Consumption of a part object means change of the aggregate object, therefore **Assembly Line** is affected by **Part Assembling** and since **Parts Set** is part of **Assembling**, **Assembly Line** is affected by **Assembling**. The result of the abstraction process is shown in Figure 35



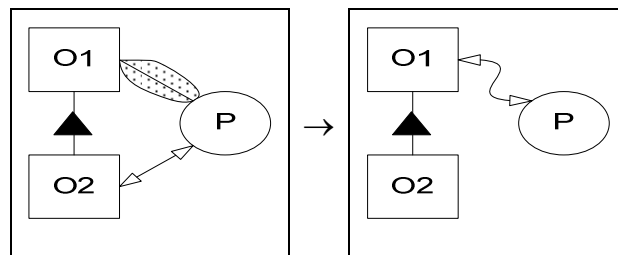
**Figure 35. OPD after abstraction**

Procedural abstraction can be done when two entities of the same persistence are connected with an aggregation-participation or an exhibition-characterization link.

generalization-specialization links and classification-instantiation links are treated differently and do not count in this abstraction process.

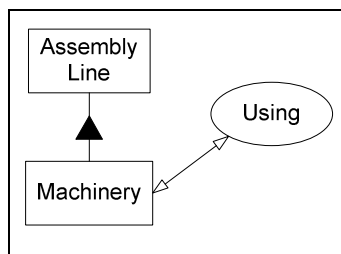
For categorization purposes, the productions are divided into two main groups depending on the kind of link that connects the two entities: aggregation-participation and exhibition-characterization. Each is further divided into two groups: object-based abstractions, in which productions are applied to objects, and process-based abstraction, in which productions are applied to processes. The productions are further divided into groups by the link that is being abstracted.

Although productions can be applied to any match that occurs in the OPD that is being abstracted, the abstraction algorithm defines a specific order, which is dictated by the **thing** that the algorithm selects in step 1.1. The match for the applied production in the abstraction process must map **thing** either to O2, if **thing** is an object, or to P2, if **thing** is a process. For example, suppose the source diagram is the diagram shown in Figure 34 and the thing selected by the current iteration of the abstraction algorithm is **Machinery**. A possible production that can be applied to the diagram is Production 9.2.1, which is shown in Figure 36. Production 9.2.1



**Figure 36. Production 9.2.1 – Promotion of Part Effect to Aggregate Effect**

As stated above, the selected thing must be mapped to O2, that is, **Machinery** must be mapped to O2. Continuing from this, the only available match for the diagram will map **Assembly Line** to O1 and **Using** to P. The match is shown in Figure 37.



**Figure 37. Match in Assembly Line OPD for Production 9.2.1**

### 5.3.2.1 Aggregation-Participation Based Procedural Link

#### Abstraction

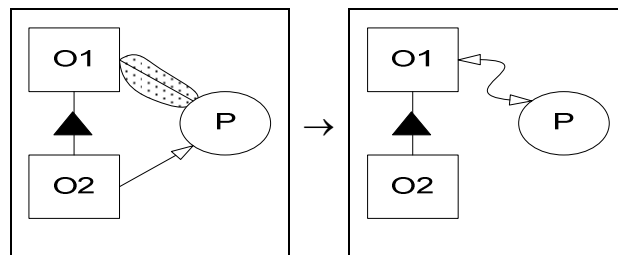
The following productions define the abstraction of procedural links that start at part-things. They are divided into subcategories for classification purposes: Object-based productions—both the part and the aggregate are objects, Process-based productions—both the part and the aggregate are processes, and Mixed-productions—where one of the things is an object and the other is a process.

Each production is shown as a graph grammar production. The semantic rationale behind the production is explained below its graphical definition.

#### Production 9 Aggregation Object-Based Productions

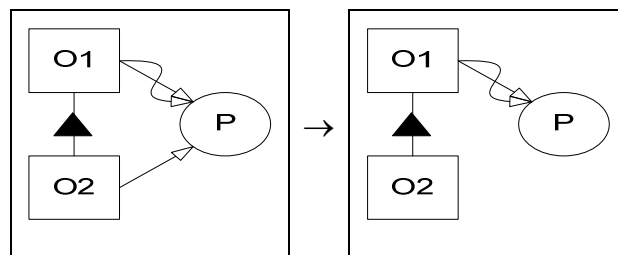
##### Production 9.1 Consumption Link Abstraction Productions

##### Production 9.1.1 Promotion of Part Consumption to Aggregate Effect



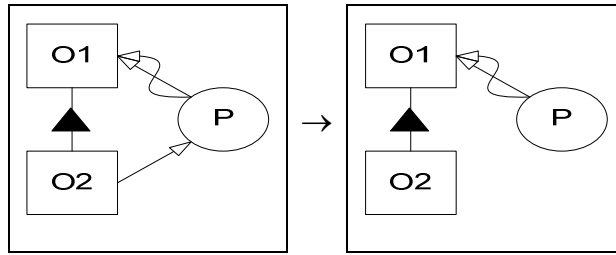
Consumption of a part object affects the aggregate object. Since in the diagram there is no specified relation between O1 and P, the fact that it is affected by P must be added to the diagram. After this is done, the link between O2 and P is removed.

##### Production 9.1.2 Part Consumption Removal via Consumption



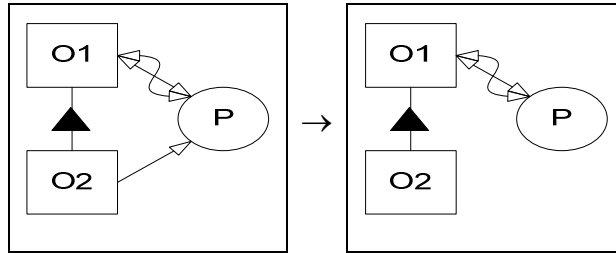
Consumption of an aggregate means optional but not mandatory consumption of some subset of its parts. Because of this, the link between O2 and P is abstracted by the link between O1 and P, and is removed.

##### Production 9.1.3 Part Consumption Removal via Result



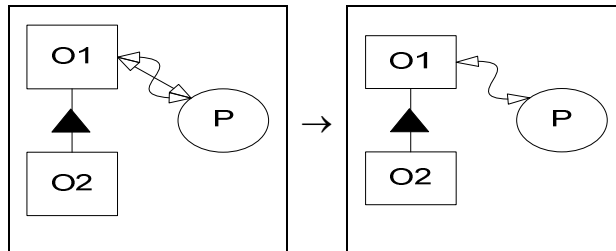
Yielding an aggregate means changing it, and might involve consumption of a part of it, therefore the link between O2 and P is removed.

**Production 9.1.4** Part Consumption Removal via Effect



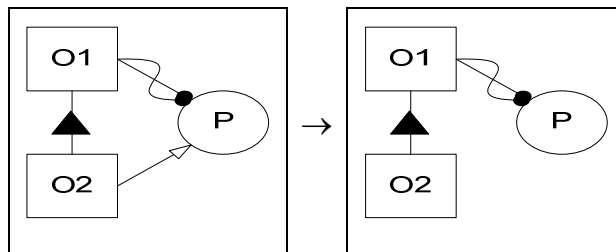
Effect on an aggregate can abstract consumption of a part, therefore the consumption link between O2 and P is removed.

**Production 9.1.5** Part Consumption Removal while Upgrading Instrument to Effect



Consumption of a part means change to the aggregate. Since the link between O1 and P was created by the abstraction algorithm, its kind can be changed as long as its new kind expands the meaning of the original link (this is referred in this work as "upgrading"). In this case, the instrument link can be upgraded to an effect, and the link between O2 and P is then removed.

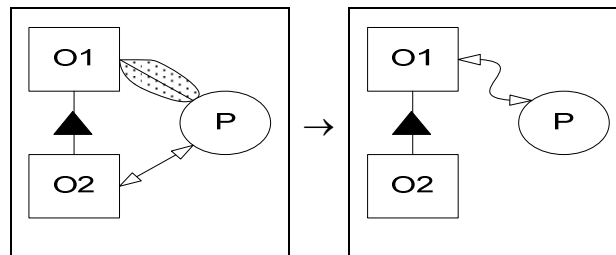
**Production 9.1.6** Part Consumption Removal via Agent



An agent link defines that an object is the human (or group of humans) handler of a process. While not specifically stated, an agent may also be affected by the process. Since consumption of a part is abstracted by effect to its aggregate, and the agent link abstracts effect, the link between O2 and P is removed.

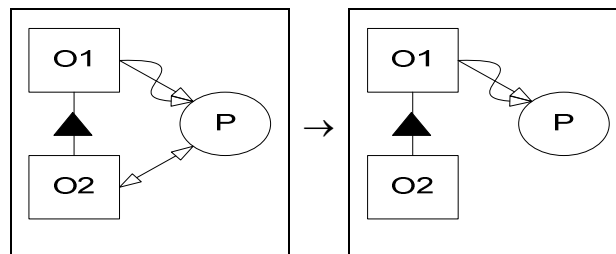
## Production 9.2 Effect Link Abstraction Productions

### Production 9.2.1 Promotion of Part Effect to Aggregate Effect



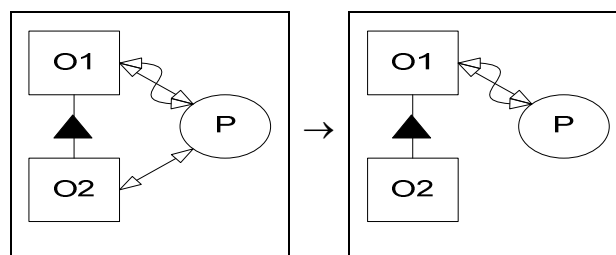
Effect caused to a part object means effect to the aggregate object. Since in the diagram there is no specified relation between O1 and P, the fact that it is affected by P must be added to the diagram. After this is done, the link between O2 and P is removed.

### Production 9.2.2 Part Effect Removal via Consumption



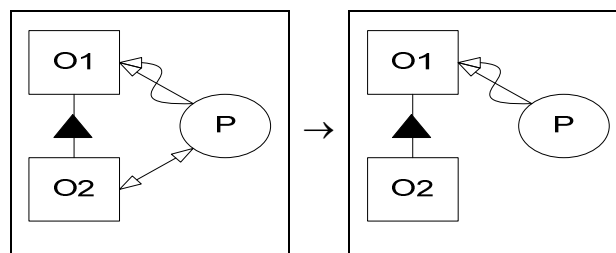
Effect on a part object is abstracted by effect on the aggregate of the object. Furthermore, effect on an object can be abstracted by consumption. Therefore the link between O2 and P is removed.

### Production 9.2.3 Part Effect Removal via Effect



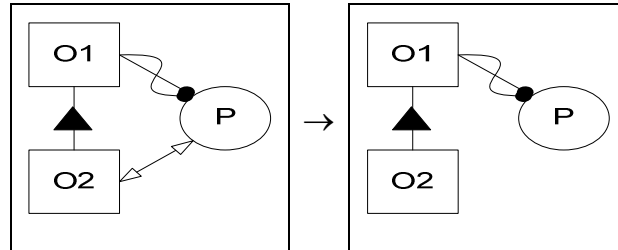
Effect caused to a part object is trivially abstracted as effect caused to the aggregate. Therefore the link between O2 and P is removed.

### Production 9.2.4 Part Effect Removal via Result



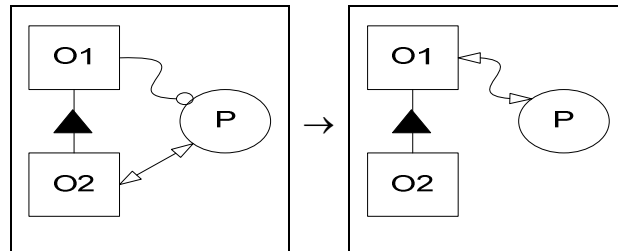
Effect on a part object is abstracted by effect on the aggregate of the object. Furthermore, effect on an object can be abstracted by result. Therefore the link between O2 and P is removed.

**Production 9.2.5 Part Effect Removal via Agent**



As stated before, and agent link may also includes change to the agent object. Effect on a part object is abstracted by effect on its aggregate and this is abstracted by the agent link. Therefore the link between O2 and P is removed.

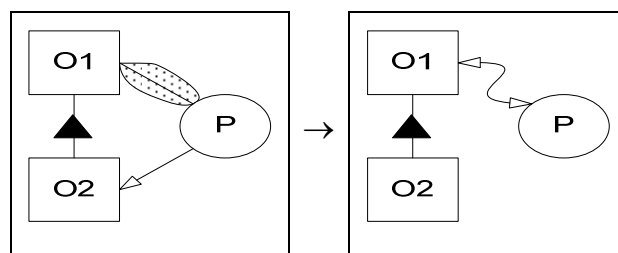
**Production 9.2.6 Part Effect Removal while Upgrading Instrument to Effect**



Effect on a part object is abstracted by effect on the aggregate. Since the relation between O1 and P was created by the algorithm, it is upgraded to an effect link for it to abstract the effect on the part object. After this, the link between O2 and P is removed.

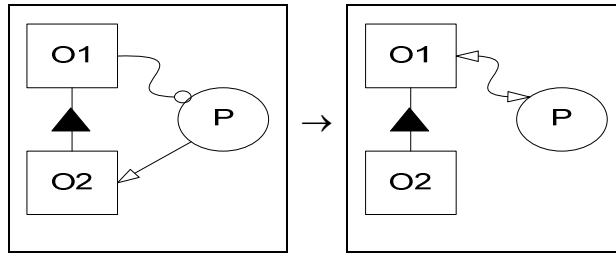
**Production 9.3 Result Link Abstraction Productions**

**Production 9.3.1 Promotion of Part Result to Aggregate Effect**



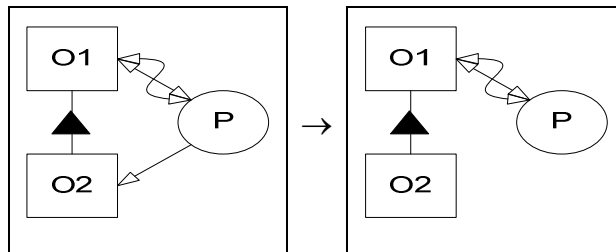
Result of a part object means effect on the aggregate object. Since in the diagram there is no specified relation between O1 and P, the fact that it is affected by P must be added to the diagram. After this is done, the link between O2 and P is removed.

**Production 9.3.2 Part Result Removal While Upgrading Instrument to Effect**



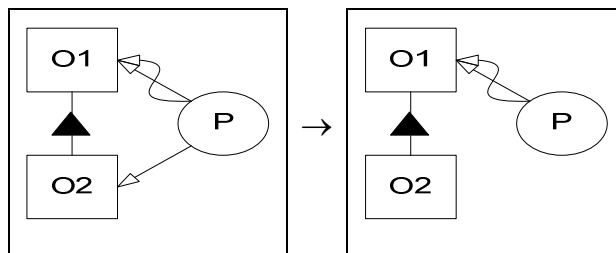
Result of a part object means effect to the aggregate object. Because the link between O1 and P was created by the algorithm, it is upgraded to an effect link so it correctly abstracts the result link. After this is done, the link between O2 and P is removed.

**Production 9.3.3 Part Result Removal via Effect**



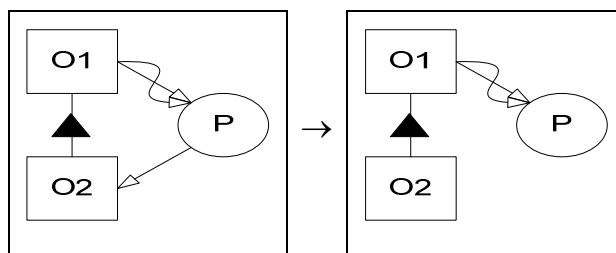
As stated before, result of a part object means change to its aggregate, therefore the link between O2 and P is correctly abstracted and is removed.

**Production 9.3.4 Part Result Removal via Result**



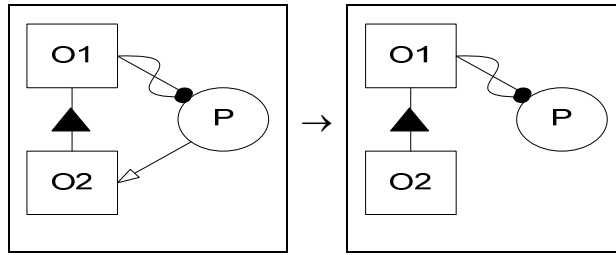
Result of a part is trivially abstracted by result of the aggregate object. Therefore the link between O2 and P is removed.

**Production 9.3.5 Part Result Removal via Consumption**



Consuming an aggregate means change to it, the same as consumption of a part of it therefore the link between O2 and P is removed because it is correctly abstracted.

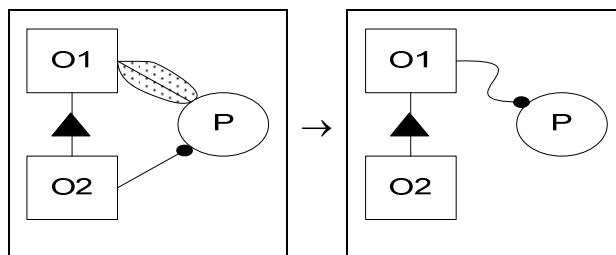
**Production 9.3.6 Part Result Removal via Agent**



Result of a part object means change to its aggregate. Since an agent link also represents optional change to the agent, the result link between O2 and P is correctly abstracted and is removed.

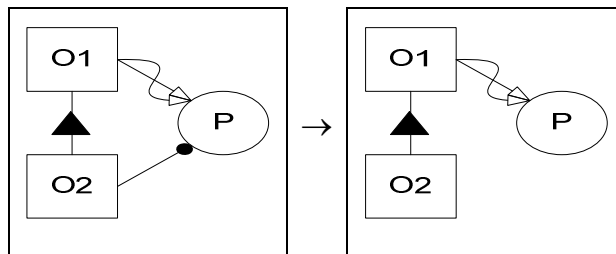
**Production 9.4 Agent Link Abstraction Productions**

**Production 9.4.1 Promotion of Part Agent to Aggregate Agent**



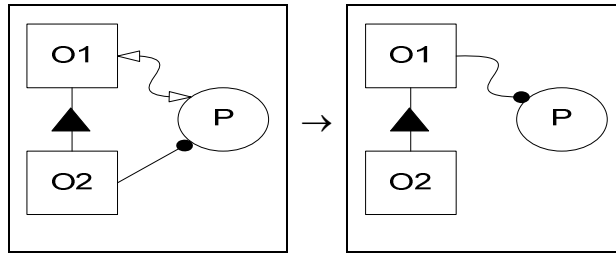
If a part object is an agent of a process, the aggregate object is also an agent, otherwise information is lost. Since there is no link between O1 and P, this link is added and the link between O2 and P is removed.

**Production 9.4.2 Part Agent Removal via Consumption**



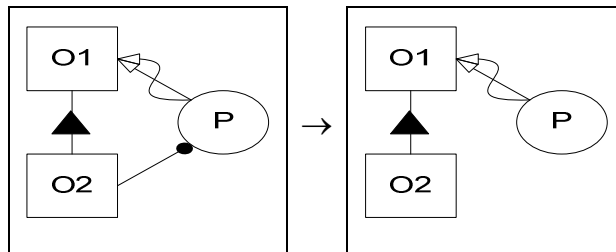
This is the first production where some information is lost without being abstracted. Semantically, if a part is an agent of a process, and aggregate agent abstracts this. But in this diagram the aggregate is consumed by the process. Because consumption is a very strong operation, changing it will create a greater loss of information. Therefore after some discussions, declaring the source diagram an illegal construct seemed too drastic and it was decided that the agent link is abstracted by the consumption link, even though the best possibility would be to allow duplicate links between an object and a process in this case, something that is not currently allowed in OPM. Semantically, this can happen for example if a person is part of a department and the department is then dismantled.

**Production 9.4.3 Part Agent Removal while Upgrading Effect to Agent**



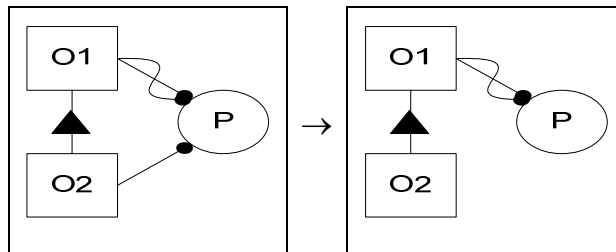
As noted, and agent also includes effect, but effect does not abstract agent, therefore the link between O1 and P is upgraded and after this the link between O2 and P is removed.

**Production 9.4.4** Part Agent Removal via Result



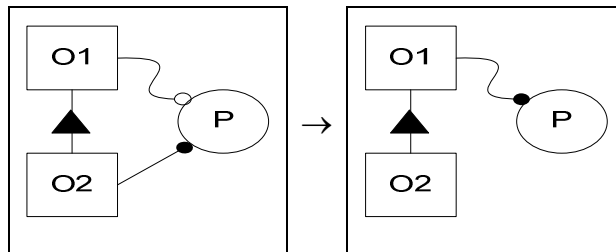
Like Production 9.4.2, this production is also controversial, but as was done there, the decision was done to allow this mixture while losing some of the information in the OPD.

**Production 9.4.5** Part Agent Removal via Agent



An agent link in the aggregate trivially abstracts agent link or its parts, therefore the link between O2 and P is removed.

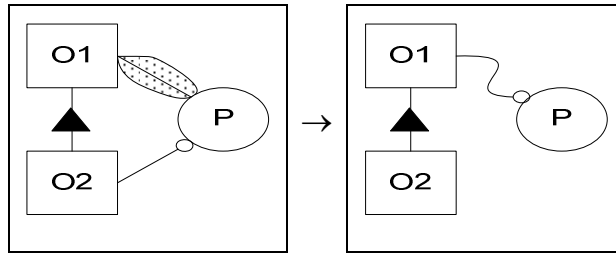
**Production 9.4.6** Part Agent Removal while Upgrading Instrument to Agent



The instrument link in the aggregate does not abstract the agent link of its parts, but since it was created by the abstraction algorithm, it can be upgraded to an agent link, and after this the link between O2 and P is removed.

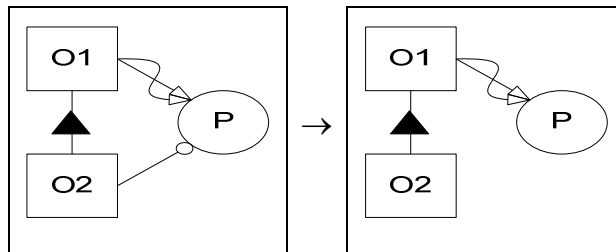
**Production 9.5** Instrument Link Abstraction Productions

**Production 9.5.1** Promotion of Part Instrument to Aggregate Instrument



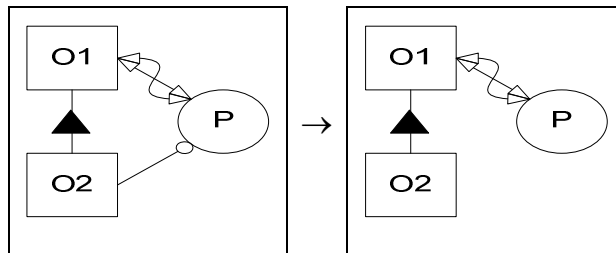
If a part object is an instrument of a process, the aggregate object is also an instrument, otherwise information is lost. Since there is no link between O1 and P, this link is added and the link between O2 and P is removed.

**Production 9.5.2** Part Instrument Removal via Consumption



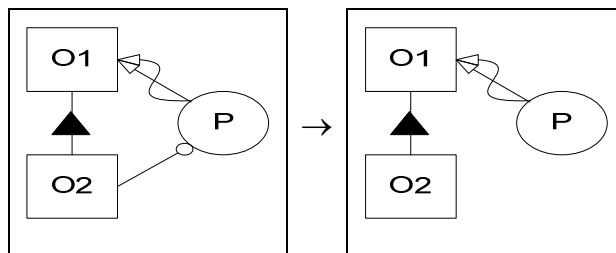
Consumption of an aggregate abstracts the requirement of its parts (instruments) therefore the link between O2 and P is removed.

**Production 9.5.3** Part Instrument Removal via Effect



Effect caused to an aggregate abstracts the requirement of its parts (instruments) therefore the link between O2 and P is removed.

**Production 9.5.4** Part Instrument Removal via Result

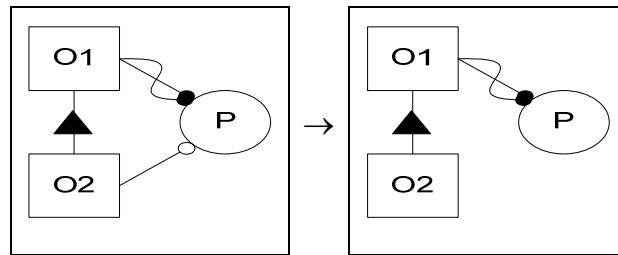


Result of an aggregate abstract requirement of its parts (instruments) therefore the link between O2 and P can be removed.

This production may seem strange since it shows that a process uses part of an object at the same time that it creates it. But in OPM the parts of a thing are independent of the thing itself, so this can occur. Furthermore, since this is an abstraction of a larger diagram, what may happen is that O1 is created by a sub-process of P and afterwards

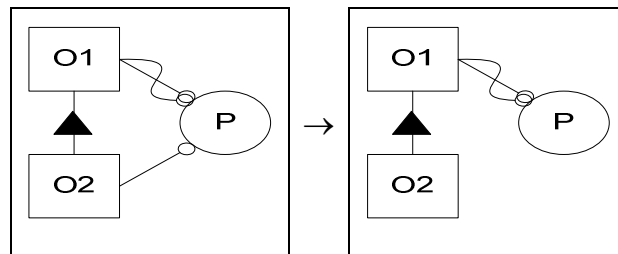
another sub-process of P uses O2 as an instrument. Once again, the abstraction process causes loss of information.

**Production 9.5.5 Part Instrument Removal via Agent**



An aggregate that is agent to a process abstracts that a part of it is instrument to the process, therefore the link between O2 and P is removed.

**Production 9.5.6 Part Instrument Removal via Instrument**

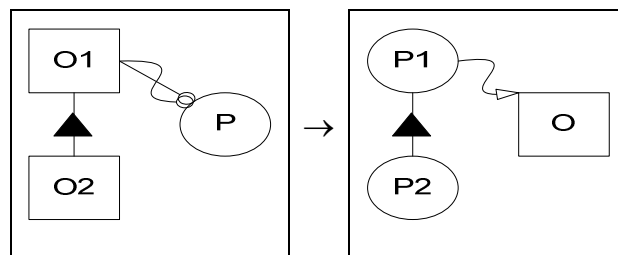


An aggregate that is instrument to a process trivially abstracts that a part of it is also instrument to the process, therefore the link between O2 and P is removed.

**Production 10 Aggregation Process-Based Productions**

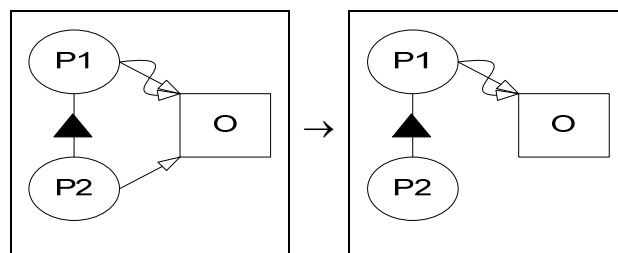
**Production 10.1 Result Link Abstraction Productions**

**Production 10.1.1 Promotion of Part Result to Aggregate Result**



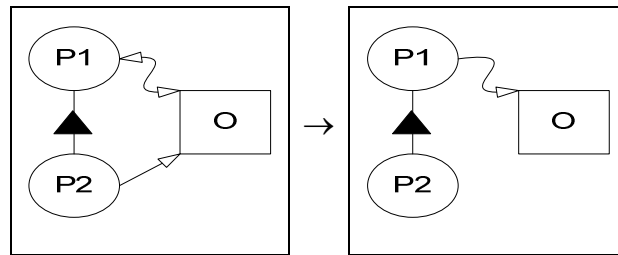
If a part of a process yields an object, the correct abstraction in the aggregate is also to yield the object. Since there is no link between P1 and O, this link is added and the link between P2 and O is removed.

**Production 10.1.2 Part Result Removal via Result**



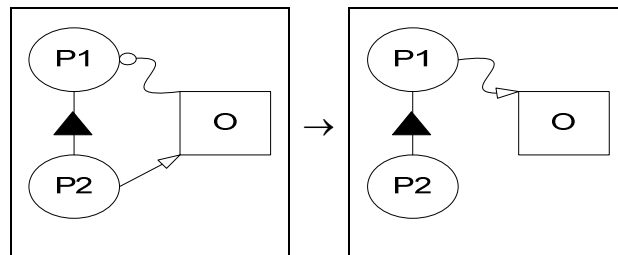
An aggregate that yields an object abstract that a part of it yields it, therefore the link between P2 and O is removed.

**Production 10.1.3** Part Result Removal while Upgrading Effect to Result



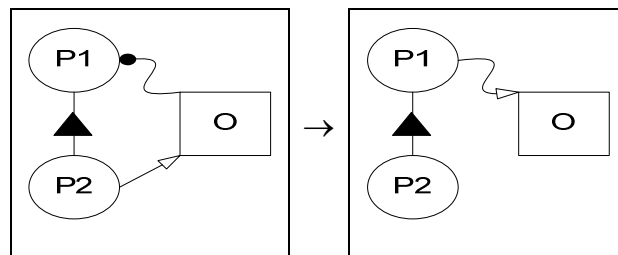
An aggregate that affects an object does not abstract that the object is yielded by a part of it, but since the link between P1 and O was created by the abstraction algorithm, it can be upgraded to a result link. After this the link between P2 and O is deleted.

**Production 10.1.4** Part Result Removal while Upgrading Instrument to Result



An aggregate that requires an instrument object does not abstract that a part of it yields the object, but since the link between P1 and O was created by the abstraction it can be upgraded to a result link. After this the link between P2 and O is deleted.

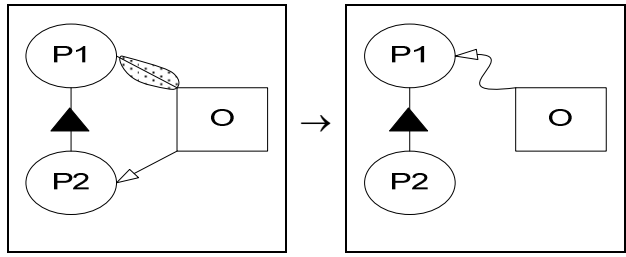
**Production 10.1.5** Part Result Removal while Upgrading Agent to Result



Like in the previous object-based productions including agent and consumption/result links, this case is also problematic. It was decided that a process that requires and agent object does not abstract that a part of the process yields the object, but that the agent link can be upgraded to a result link, this allowing the abstraction. After this the link between P2 and O is removed.

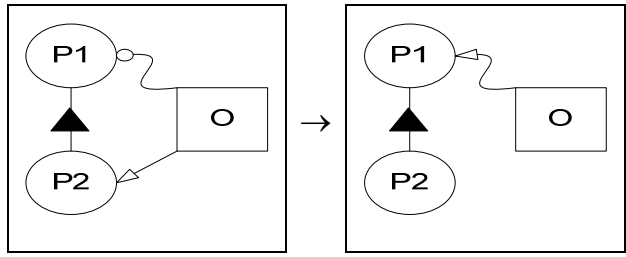
**Production 10.2** Consumption Link Abstraction Productions

**Production 10.2.1** Promotion of Part Consumption to Aggregate Consumption



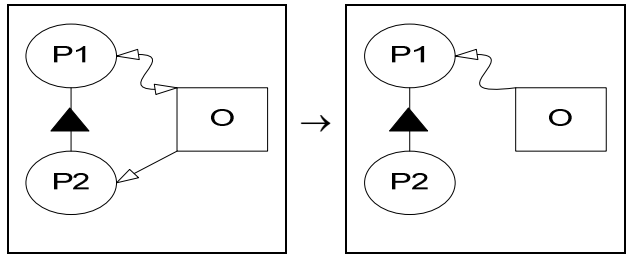
If a part of a process consumes an object, the correct abstraction in the aggregate is also to consume the object. Since there is no link between P1 and O, this link is added and the link between P2 and O is removed.

**Production 10.2.2** Part Consumption Removal while Upgrading Instrument to Consumption



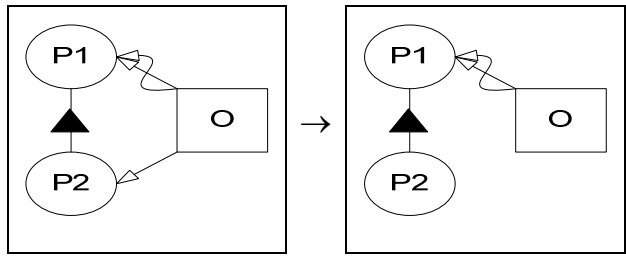
An aggregate that requires an instrument object does not abstract that a part of it consumes the process, but since the instrument link was created by the abstraction process it can be upgraded to a consumption link, and then the link between P2 and O can be removed.

**Production 10.2.3** Part Consumption Removal while Upgrading Effect to Consumption



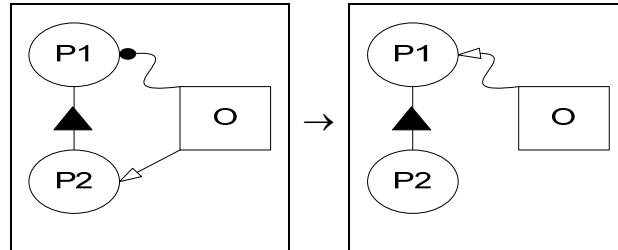
Effect caused by an aggregate does not abstract consumption by a part, but since the link was created by the abstraction algorithm, it can be upgraded to a consumption link and then the link between P2 and O can be removed.

**Production 10.2.4** Part Consumption Removal via Consumption



Consumption by an aggregate trivially abstract consumption by a part of it, therefore the link between P2 and O can be removed.

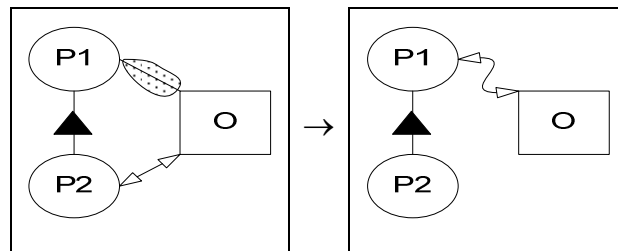
**Production 10.2.5** Part Consumption Removal while Upgrading Agent to Consumption



Like in the previous productions including agent and consumption/result links, this case is also problematic. It was decided that a process that requires and agent object does not abstract that a part of the process consumes the object, but that the agent link can be upgraded to a result link, this allowing the abstraction. After this the link between P2 and O is removed.

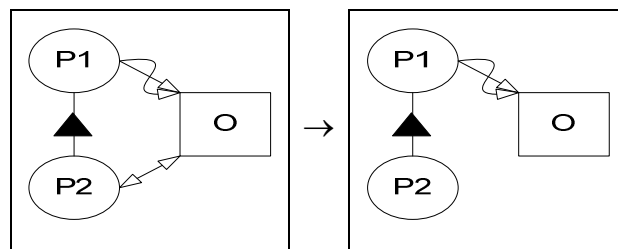
**Production 10.3** Effect Link Abstraction Productions

**Production 10.3.1** Promotion of Part Effect to Aggregate Effect



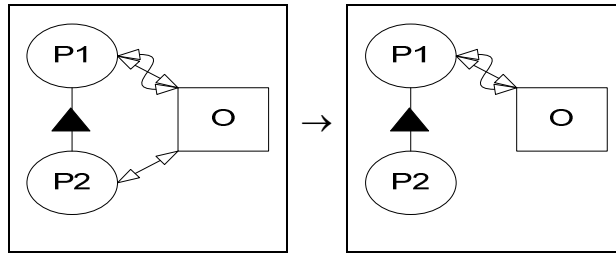
If a part of a process affects an object, the correct abstraction in the aggregate is also to affect the object. Since there is no link between P1 and O, this link is added and the link between P2 and O is removed.

**Production 10.3.2** Part Effect Removal via Result



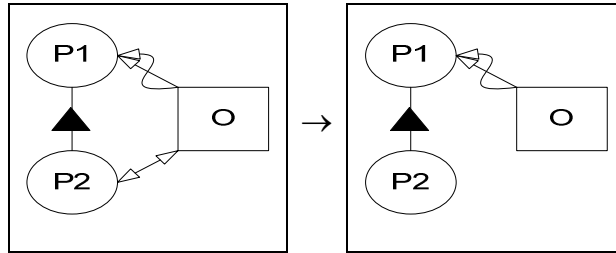
An aggregate that yields an object correctly abstract that a part of it affects the object, so the link between P2 and O is abstracted and can be removed.

**Production 10.3.3** Part Effect Removal via Effect



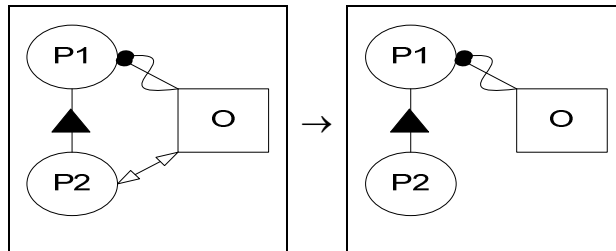
Effect caused by an aggregate trivially abstracts effect caused by a part of it, therefore the link between P2 and O can be removed.

**Production 10.3.4** Part Effect Removal via Consumption



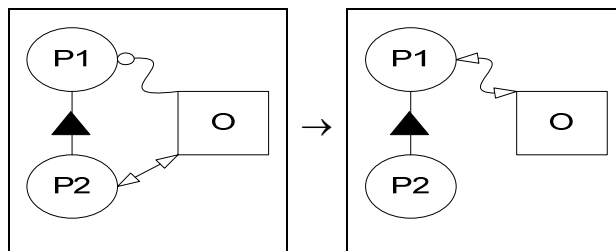
An aggregate process that consumes an object correctly abstract that a part of it also affects the object, so the link between P2 and O is abstracted and can be removed.

**Production 10.3.5** Part Effect Removal via Agent



As stated before, an agent link implies optional effect to the object, therefore the aggregate process P1 implies optional effect to O, which abstract effect caused by P2, so the link between P2 and O is removed.

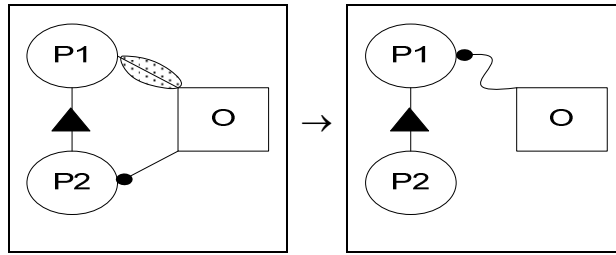
**Production 10.3.6** Part Effect Removal while Upgrading Instrument to Effect



An instrument link in the aggregate does not abstract effect caused by a part, but since the link was created by the algorithm, it can be upgraded to an effect link. After this the link between P2 and O is removed.

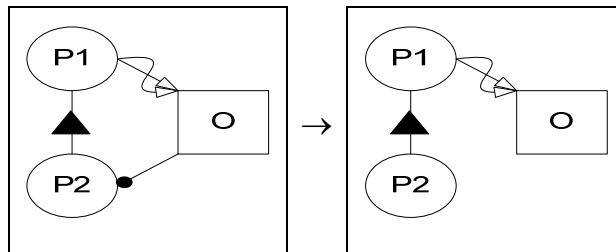
**Production 10.4** Agent Link Abstraction Productions

**Production 10.4.1** Promotion of Part Agent to Aggregate Agent



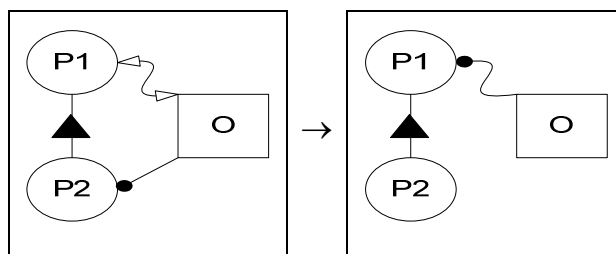
If an object is an agent of a part process, it is also required as an agent by the parent of the process. Since this is not stated by the diagram, the link is added and the link between P2 and O can be removed.

**Production 10.4.2** Part Agent Removal via Result



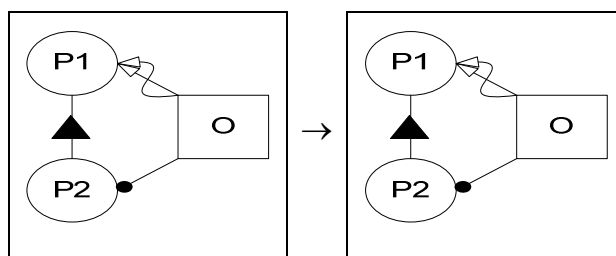
As all production involving mixes of agent and consumption/yielding, this production is not straightforward. As before, the result link is chosen as the one with more information and as such it is defined as abstracting the agent link, therefore the link between P2 and O is removed.

**Production 10.4.3** Part Agent Removal while Upgrading Effect to Agent



An effect link in the aggregate does not abstract the agent link in the part, but since the effect link was created by the abstraction process, it can be upgraded to an agent link. After this the link between P2 and O is removed.

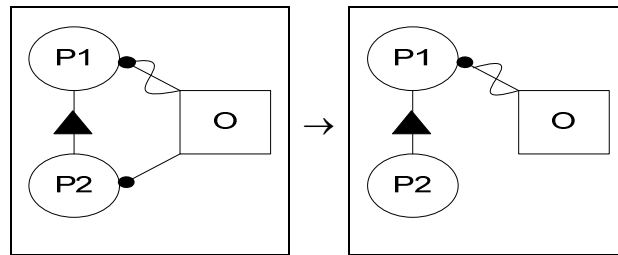
**Production 10.4.4** Part Agent Removal via Consumption



As other productions involving mixes of agent and consumption/yielding, this production is not straightforward. It was decided as before that the consumption link

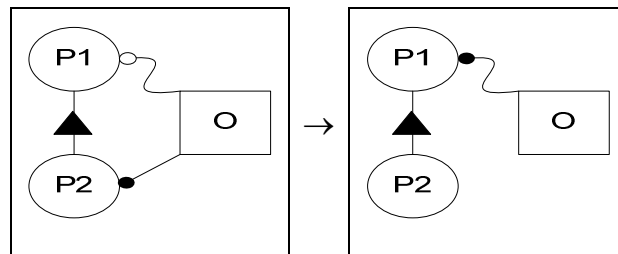
is the one with the most information and as such it abstracts the link between P2 and O, so it can be removed.

**Production 10.4.5 Part Agent Removal via Agent**



An agent link to an aggregate process trivially abstract an agent link to a part, therefore the link between P2 and O can be removed.

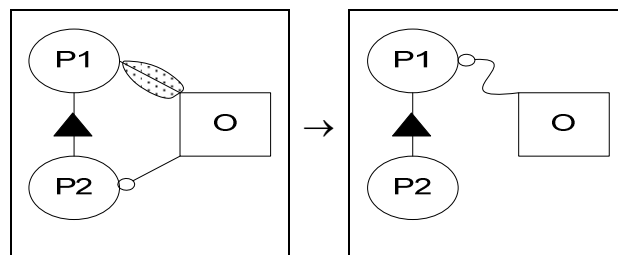
**Production 10.4.6 Part Agent Removal while Upgrading Instrument to Agent**



An instrument link to the aggregate process does not abstract the instrument link to the part process, but since the instrument link was created by the abstraction process, it can be upgraded to an agent link. After this the link between P2 and O is removed.

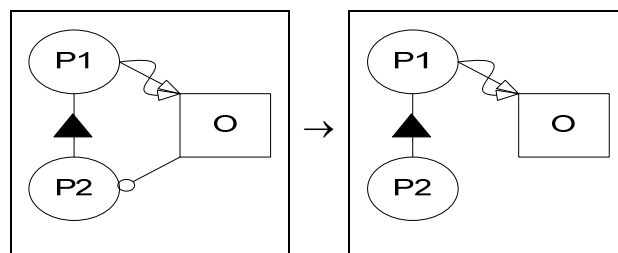
**Production 10.5 Instrument Link Abstraction Productions**

**Production 10.5.1 Promotion of Part Instrument to Aggregate Instrument**



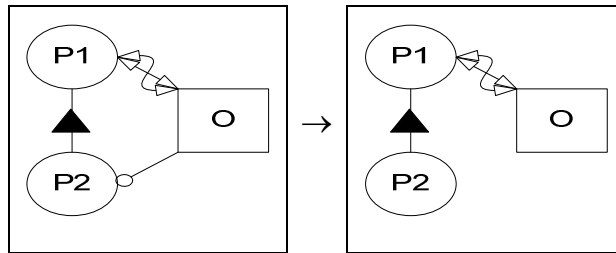
If an object is an instrument of a part process, it is also required as an instrument by the parent of the process. Since this is not stated by the diagram, the link is added and the link between P2 and O can be removed.

**Production 10.5.2 Part Instrument Removal via Result**



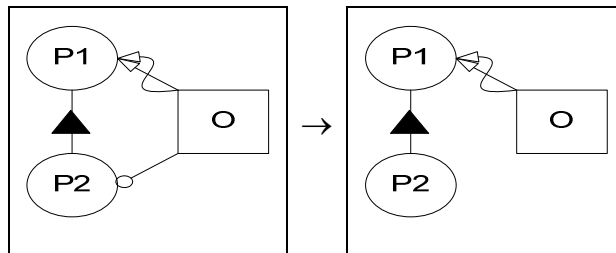
Yielding of an object by an aggregate process abstracts that a part of a process requires the object, therefore the link between P2 and O can be removed. This production could be seen as illegal, since how can a part process require an object that is created by the aggregate? Therefore the link that connects P1 and O is the abstraction of all the link of P1's parts. One part of P1 yields O while another one uses it as an instrument, so the most important link in the process is the result link. While this abstraction may be correct, the model can be built so as to require O before it is created, and this is semantic error is not checked by the abstraction algorithm.

**Production 10.5.3** Part Instrument Removal via Effect



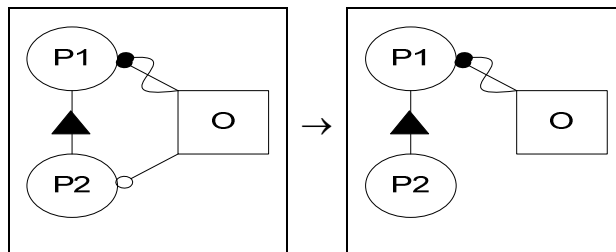
Effect caused by an aggregate process abstracts that a part of the process uses the affected object as an instrument, so the link between P2 and O is removed.

**Production 10.5.4** Part Instrument Removal via Consumption



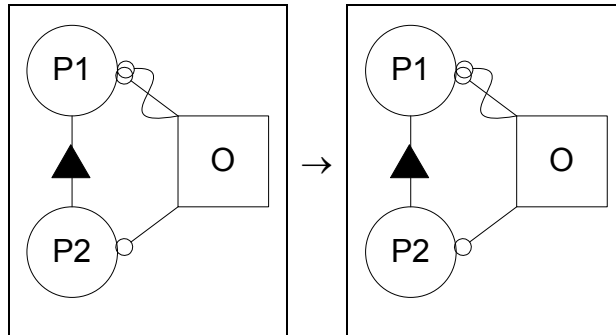
Consumption of an object by an aggregate process abstracts that a part of a process requires the object, therefore the link between P2 and O is removed.

**Production 10.5.5** Part Instrument Removal via Agent



An instrument link to the aggregate process abstract the instrument link to the part process, so the link between P2 and O is removed.

**Production 10.5.6** Part Instrument Removal via Instrument



An instrument to the aggregate process trivially abstracts instrument to a part process, so the link between P2 and O is removed.

### 5.3.2.2 Exhibition-Characterization Based Procedural Link

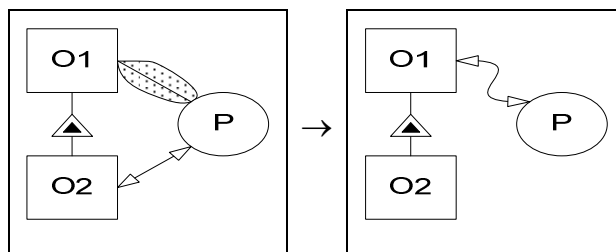
#### Abstraction Productions

The main difference between an object which is a part and an object which is an attribute is that a part exists by itself independently of the object that aggregates it. In contrast, an attribute characterizes the exhibiting object and does not exist separately from it. Therefore attributes cannot be consumed or produced and can only be affected by processes, which change their state (value), or used as enablers (agents or instruments) by processes. For all the abstraction purposes attributes are the same as part objects. The productions for attributes are also divided into groups by the link that is deleted, like those for part objects and are shown below. The rationale behind the productions here is the same as the rationale of the aggregation-participation productions, therefore it will not be repeated here.

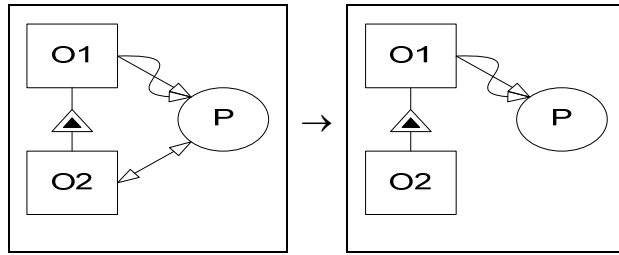
#### Production 11 Exhibition Object-Based Productions

##### Production 11.1 Effect Link Abstraction Productions

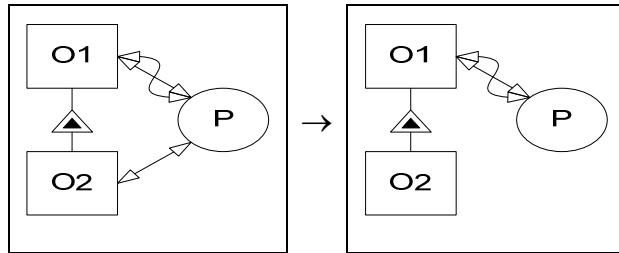
##### Production 11.1.1 Promotion of Attribute Effect to Exhibitor Effect



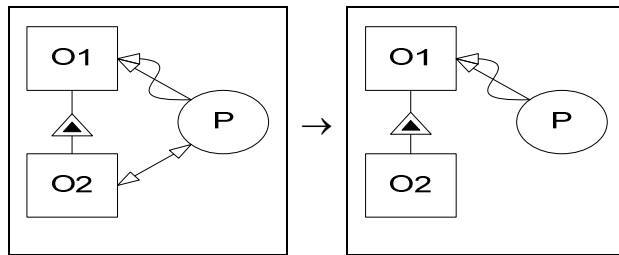
##### Production 11.1.2 Attribute Effect Removal via Consumption



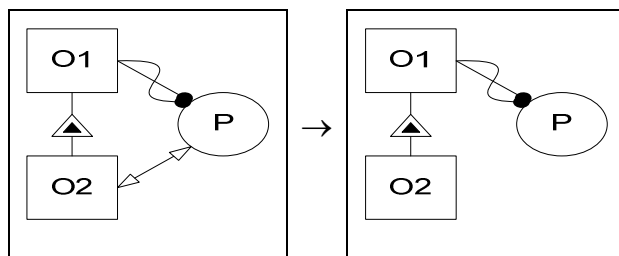
**Production 11.1.3** Attribute Effect Removal via Effect



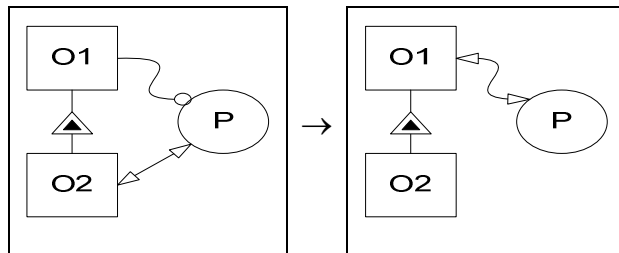
**Production 11.1.4** Attribute Effect Removal via Result



**Production 11.1.5** Attribute Effect Removal via Agent

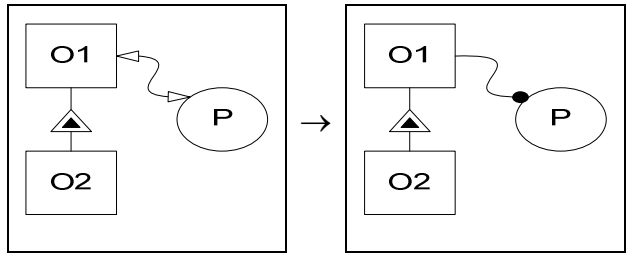


**Production 11.1.6** Attribute Effect Removal while Upgrading Instrument to Effect

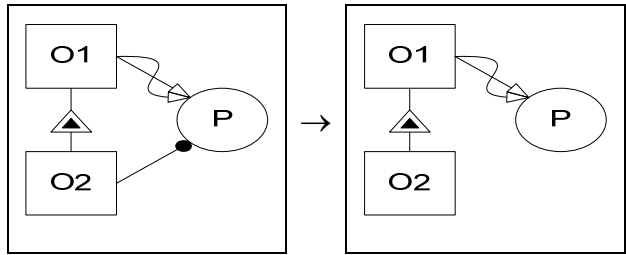


**Production 11.2** Agent Link Abstraction Productions

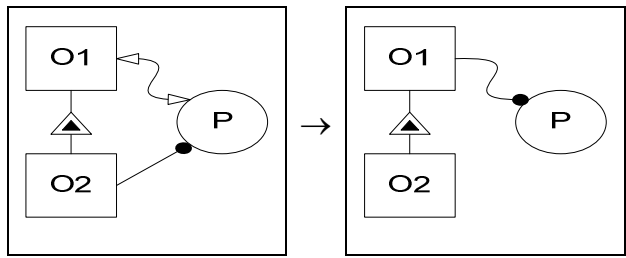
**Production 11.2.1** Promotion of Attribute Agent to Exhibitor Agent



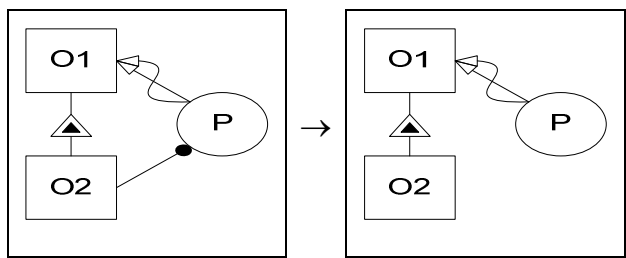
**Production 11.2.2** Attribute Agent Removal via Consumption



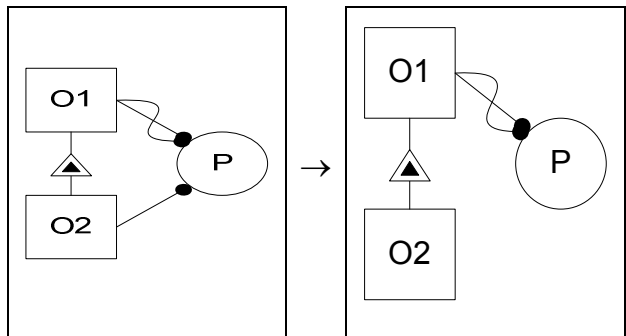
**Production 11.2.3** Attribute Agent Removal while Upgrading Effect to Agent



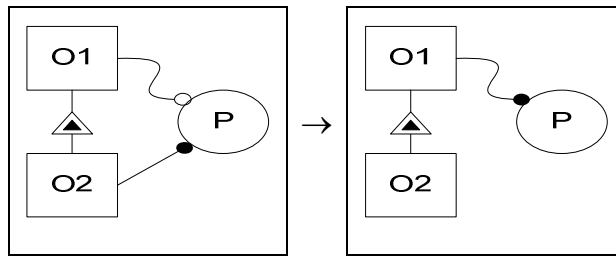
**Production 11.2.4** Attribute Agent Removal via Result



**Production 11.2.5** Attribute Agent Removal via Agent

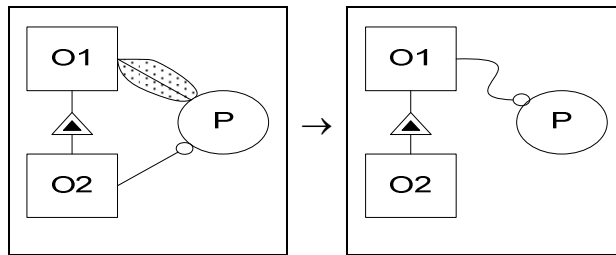


**Production 11.2.6** Attribute Agent Removal while Upgrading Instrument to Agent

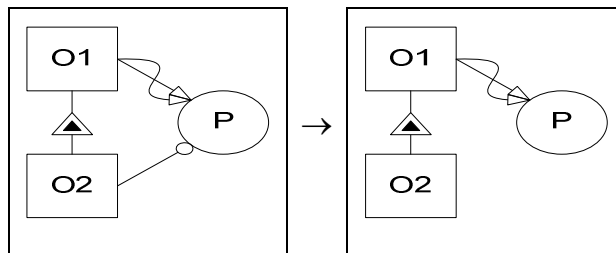


**Production 11.3** Instrument Link Abstraction Productions

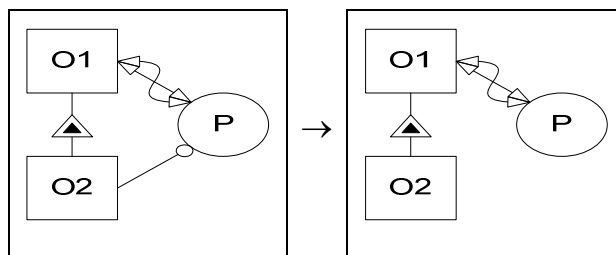
**Production 11.3.1** Promotion of Attribute Instrument to Exhibitor Instrument



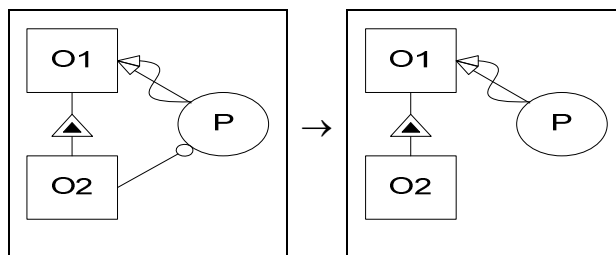
**Production 11.3.2** Attribute Instrument Removal via Consumption



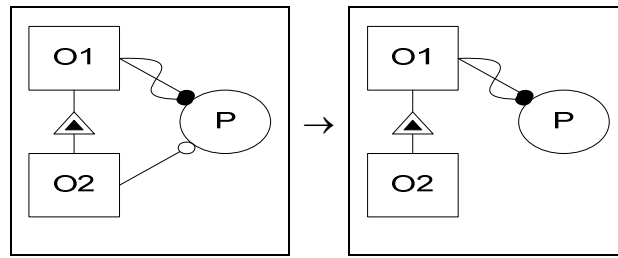
**Production 11.3.3** Attribute Instrument Removal via Effect



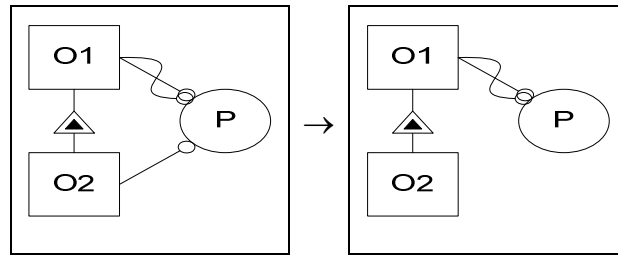
**Production 11.3.4** Attribute Instrument Removal via Result



**Production 11.3.5 Attribute Instrument Removal via Agent**



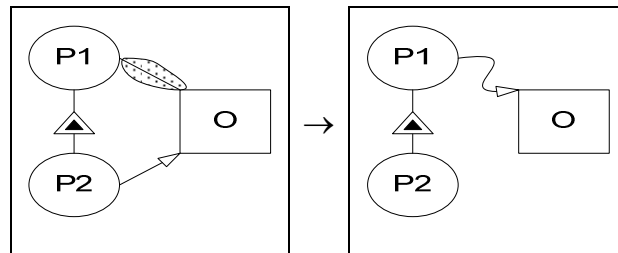
**Production 11.3.6 Attribute Instrument Removal via Instrument**



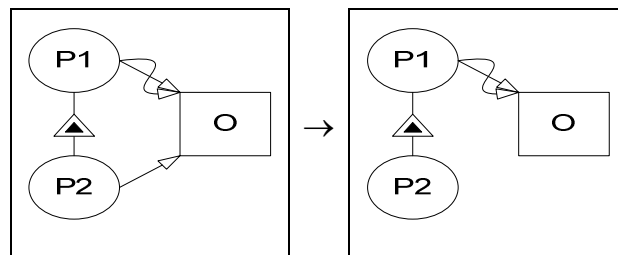
**Production 12 Exhibition Process Based Rules**

**Production 12.1 Result Link Abstraction Productions**

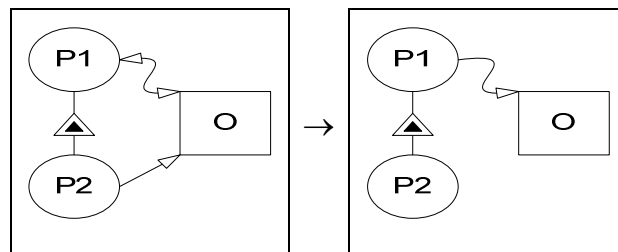
**Production 12.1.1 Promotion of Attribute Result to Exhibitor Result**



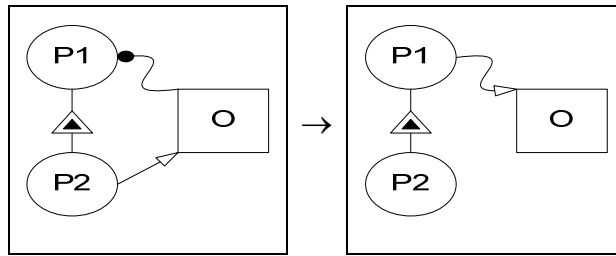
**Production 12.1.2 Attribute Result Removal via Result**



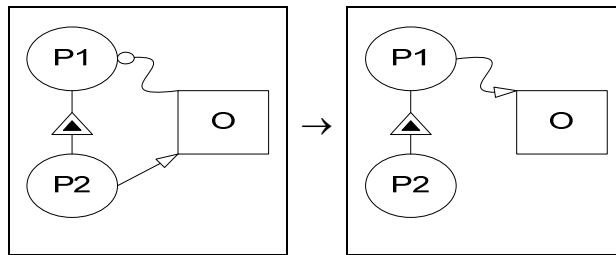
**Production 12.1.3 Attribute Result Removal while Upgrading Effect to Result**



**Production 12.1.4** Attribute Result Removal while Upgrading Agent to Result

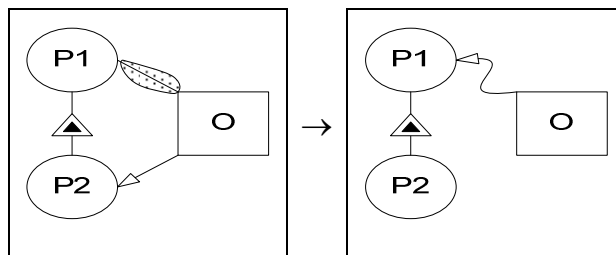


**Production 12.1.5** Attribute Result Removal while Upgrading Instrument to Result

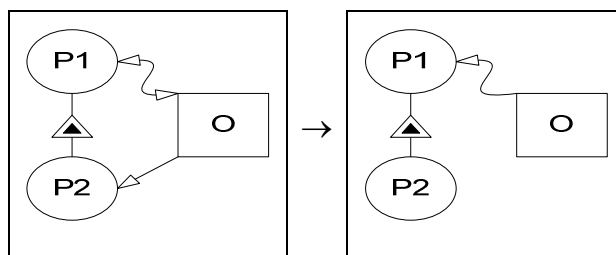


**Production 12.2** Consumption Link Abstraction Productions

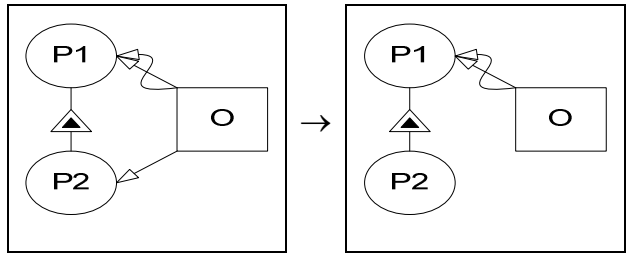
**Production 12.2.1** Promotion of Attribute Consumption to Exhibitor Consumption



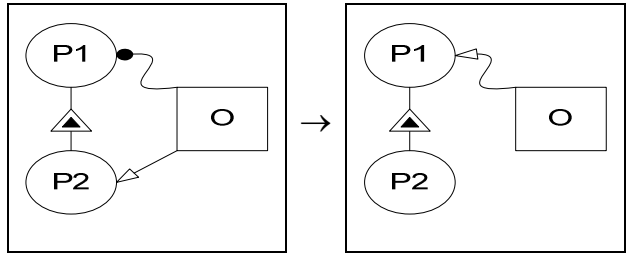
**Production 12.2.2** Attribute Consumption Removal while Upgrading Effect to Consumption



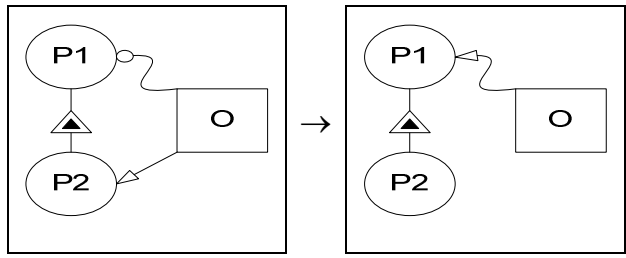
**Production 12.2.3** Attribute Consumption Removal via Consumption



**Production 12.2.4** Attribute Consumption Removal while Upgrading Agent to Consumption

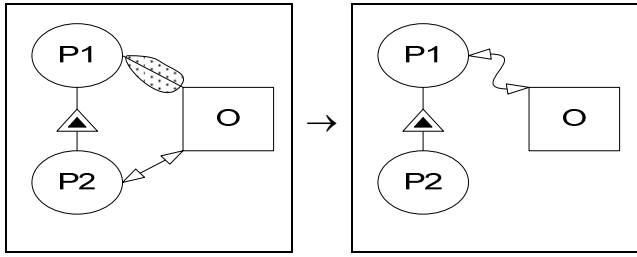


**Production 12.2.5** Attribute Consumption Removal while Upgrading Instrument to Consumption

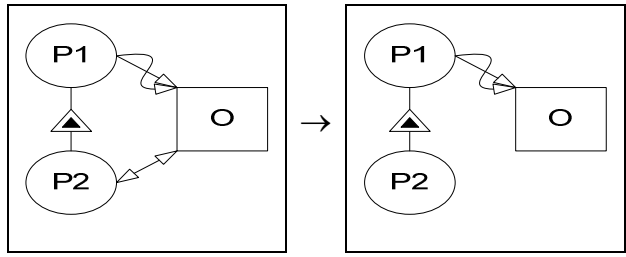


**Production 12.3** Effect Link Abstraction Productions

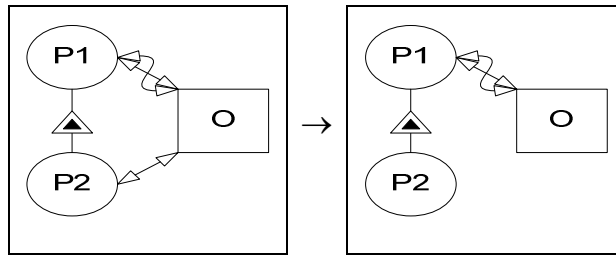
**Production 12.3.1** Promotion of Attribute Effect to Exhibitor Effect



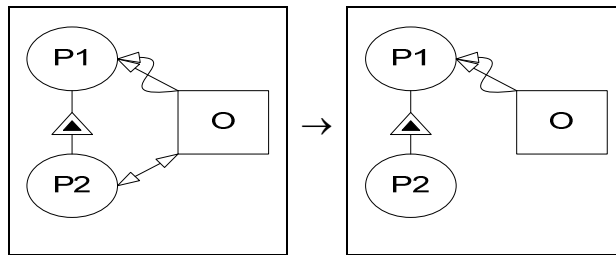
**Production 12.3.2** Attribute Effect Removal via Result



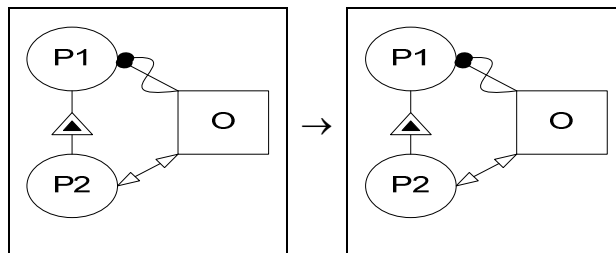
**Production 12.3.3 Attribute Effect Removal via Effect**



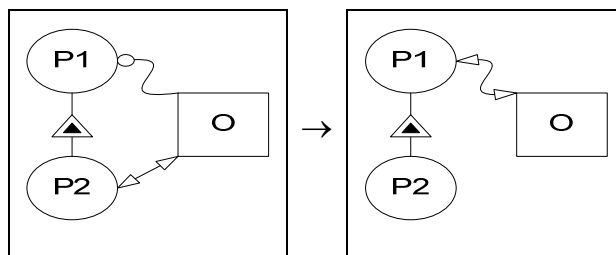
**Production 12.3.4 Attribute Effect Removal via Consumption**



**Production 12.3.5 Attribute Effect Removal via Agent**

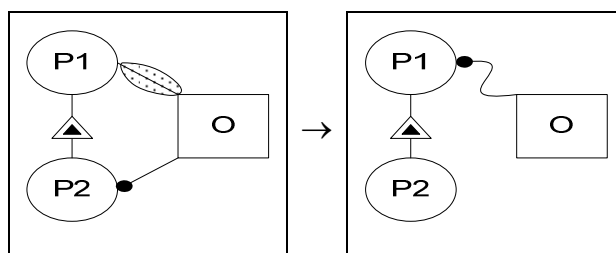


**Production 12.3.6 Attribute Effect Removal while Upgrading Instrument to Effect**

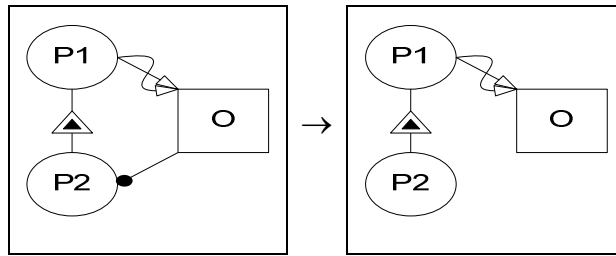


**Production 12.4 Agent Link Abstraction Productions**

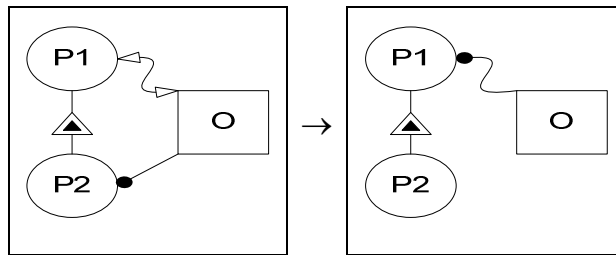
**Production 12.4.1 Promotion of Attribute Agent to Aggregate Agent**



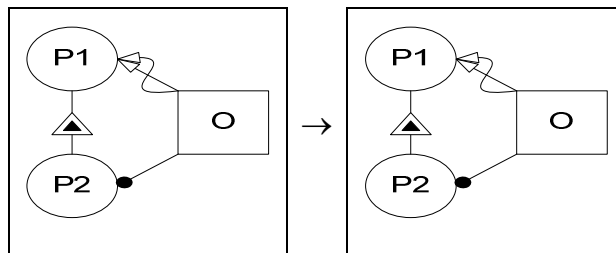
**Production 12.4.2** Attribute Agent Removal via Result



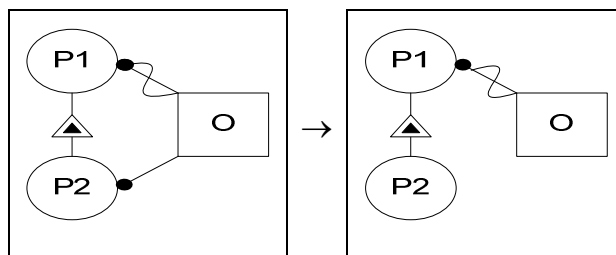
**Production 12.4.3** Attribute Agent Removal while Upgrading Effect to Agent



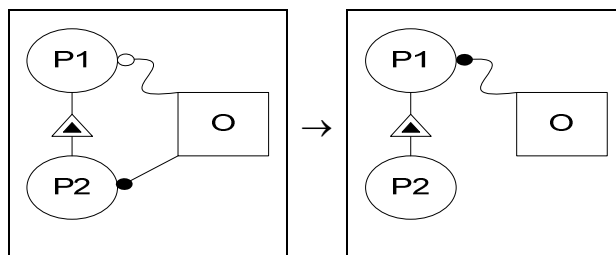
**Production 12.4.4** Attribute Agent Removal via Consumption



**Production 12.4.5** Attribute Agent Removal via Agent

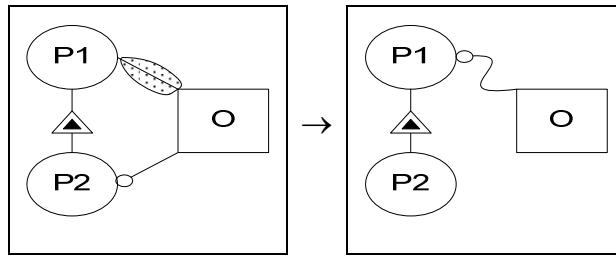


**Production 12.4.6** Attribute Agent Removal while Upgrading Instrument to Agent

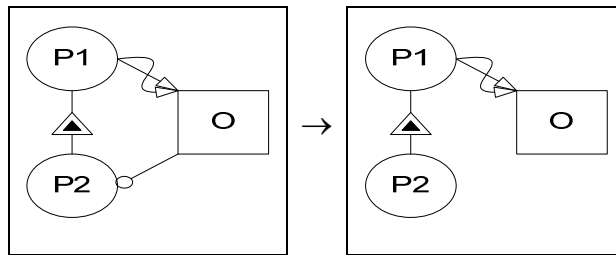


**Production 12.5 Instrument Link Abstraction Productions**

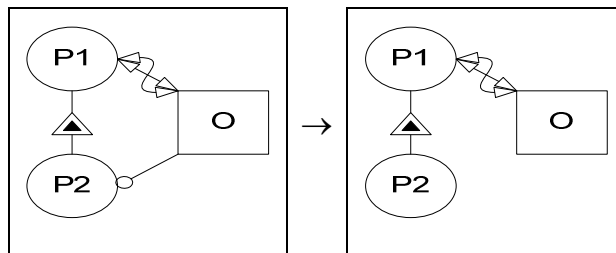
**Production 12.5.1 Promotion of Attribute Instrument to Exhibitor Instrument**



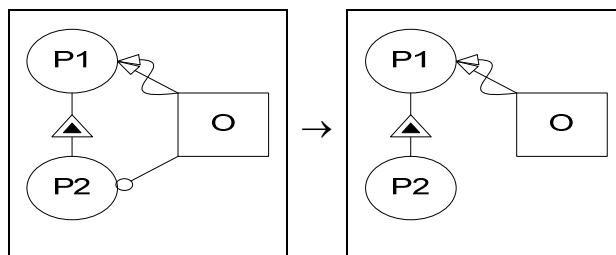
**Production 12.5.2 Attribute Instrument Removal via Result**



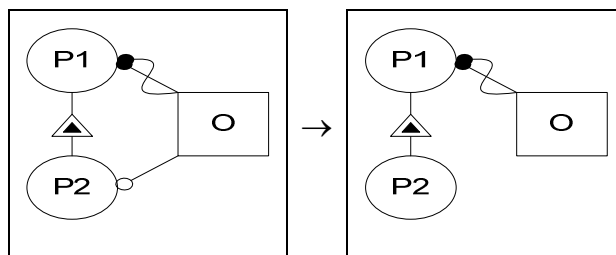
**Production 12.5.3 Attribute Instrument Removal via Effect**



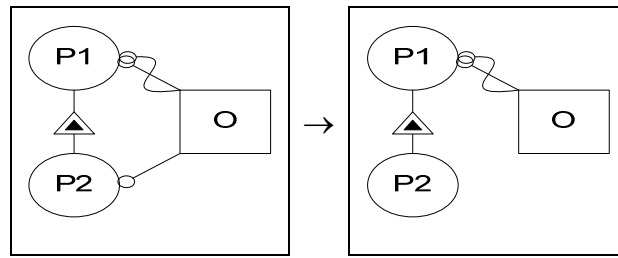
**Production 12.5.4 Attribute Instrument Removal via Consumption**



**Production 12.5.5 Attribute Instrument Removal via Agent**



**Production 12.5.6** Attribute Instrument Removal via Instrument

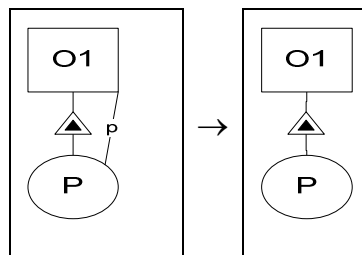


**5.3.2.3 Mixed Thing Based Procedural Link Abstraction Productions**

Structural relations are not limited to relations between thing of the same kind, but can also exist between different kinds of things – objects and processes. The abstraction of these relations is treated here using graph grammar productions. The productions are divided into two kinds: object-to-process productions (where the object is the parent of the structural relation) and process-to-object productions (where the process is the parent of the structural relation).

**Production 13** Object-to-Process Productions

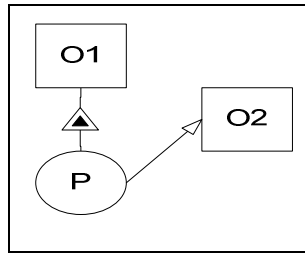
**Production 13.1** Object Exhibiting Process Procedural Link Abstraction Production



The only valid relation between an object and a process where the object is the parent of the relation is the exhibition-characterization relation. In this case the process is called the operation of the object. This relation can only be abstracted if the exhibited process is connected to an object which is itself part of the parent object.

A procedural link that links  $P$  with its parent  $O1$ , (the exhibitor of  $P$ ) can be abstracted as shown above. This production can be used because all of the links connected to the components of  $O1$  are abstracted and become connected to  $O1$ . The production takes any procedural link and removes it from the model, when the final goal is to abstract  $P$  into  $O1$ .

If the process is connected to an unrelated object, it cannot be abstracted. This can be seen in the example on Figure 38.

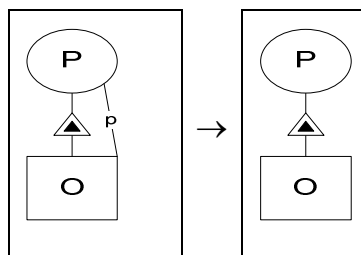


**Figure 38. Object exhibiting process**

O2 is produced by P, and O1 exhibits P, and there is no structural relation that makes O2 a part of O1.

**Production 14** Process-to-Object Productions

**Production 14.1** Process Exhibiting Object Procedural Link Abstraction Production

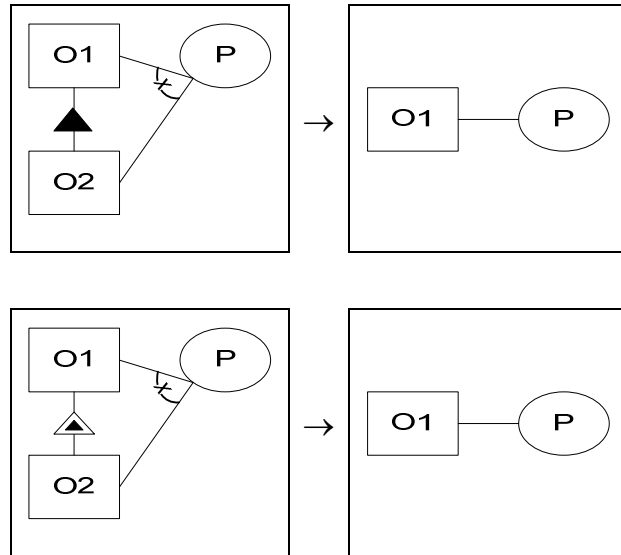


An object exhibited by a process is an attribute of the process. Because of this, all operations done on the process by the object are abstracted by the process itself and are therefore removed from the diagram.

**5.3.2.4 OR and XOR conditions**

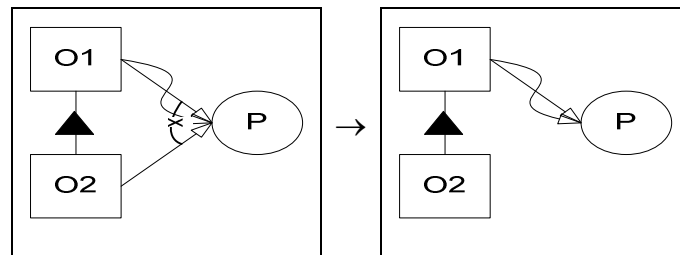
OR and XOR conditions are additional constructs that can occur at the end of a procedural link and denote that only one of the procedural links need to occur (XOR compels that only one of them occur while OR may accept both). The treatment of these conditions is identical for all kinds of links therefore it is shown here only once, to refrain from repeating all the abstraction productions for each of these conditions. Furthermore, OR and XOR are treated exactly the same. They are therefore commonly denoted by adding x along the small arc of the OR link. There are two cases of OR and XOR condition abstraction: the condition occurs with a procedural link that starts on the parent of the abstracted link (parent OR/XOR), or it occurs with a procedural link that starts at another entity (other OR/XOR). For completeness, object-based the productions will also be shown embedded in Production 9.1.2 giving a further example. All other productions are created similarly.

**Production 15** Parent OR/XOR – Object Based Procedural Link Abstraction Productions



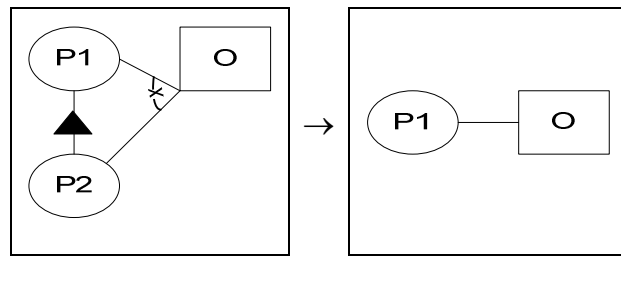
O1 consists/exhibits O2 and both are linked to P with an OR relation. Since the link that originally started at O2 is abstracted by the link that starts at O1 the OR relation becomes meaningless.

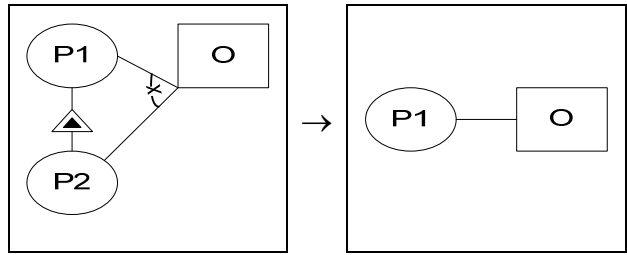
Embedding the above production into Production 9.1.2 results in the new production shown below.



**Figure 39. Example embedding of Parent OR/XOR production in an Object-Based Procedural Link Abstraction production.**

**Production 16** Parent OR/XOR – Process Based Procedural Link Abstraction Productions

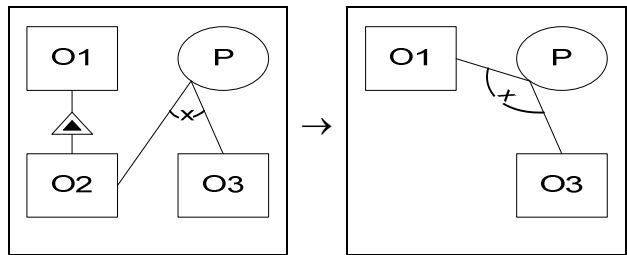
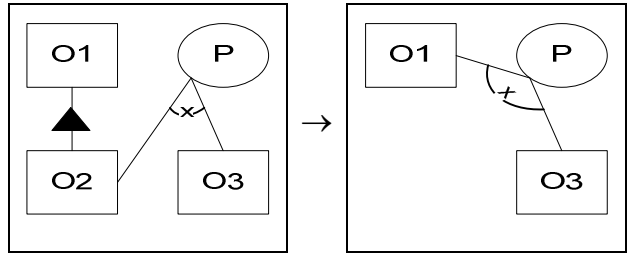




Same as above, using processes this time.

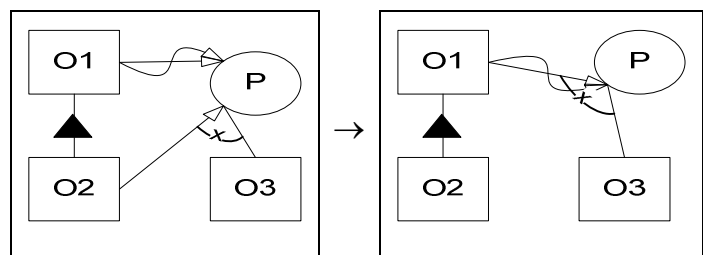
**Production 17** Other OR/XOR – Object Based Procedural Link Abstraction

Productions



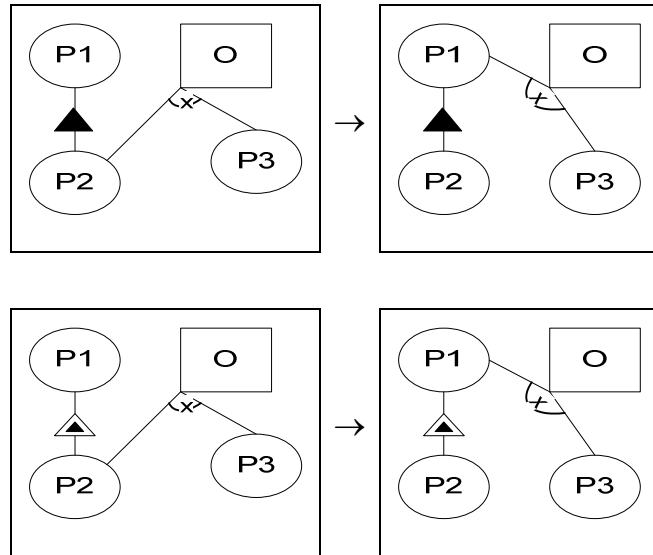
The OR relation is not with a link from the aggregate/exhibitor of O2 but from another thing O3. Since at the end of the production the link between O1 and P abstracts the original link between O2 and P then the OR relation now occurs with the link starting at O1.

Embedding the above production into Production 9.1.2 results in the new production shown below.



**Production 18** Other OR/XOR – Process Based Procedural Link Abstraction

Productions



Same as above, using processes this time.

### 5.3.3 Illegal Constructs

Illegal constructs are OPM constructs that create invalid or contradictory meaning in the OPD.

The OPD in Figure 40 shows an example of an illegal construct. Object1 is an instrument to Process1, and Object2 is consumed by Process1. This is problematic semantically because Object1 is changed by Process1. Therefore this construct is not allowed in the OPD.

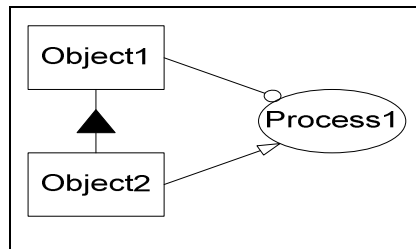
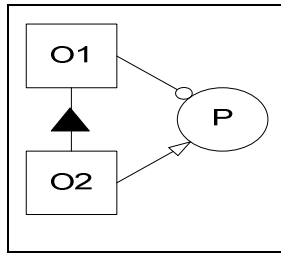


Figure 40. Illegal construct example

Following are shown all the illegal constructs in OPM that can occur in the local context of the abstraction algorithm. After each construct a brief explanation is given on the rationale behind the illegality of the construct.

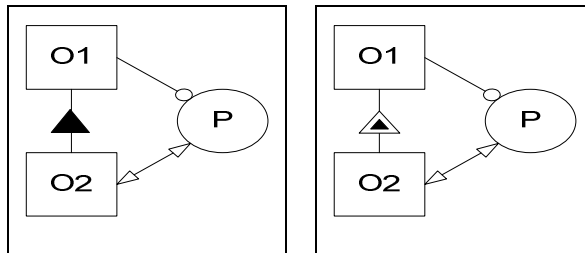
Note that there are many more illegal constructs that can occur in a non-local context, but they are irrelevant to this work since the abstraction algorithm only works in a local context. Furthermore, it can be shown that an illegal context that occurs in a non-local context will also appear in a local context in the abstraction algorithm.

#### Illegal Construct 1 Part Consumption and Aggregate Instrument



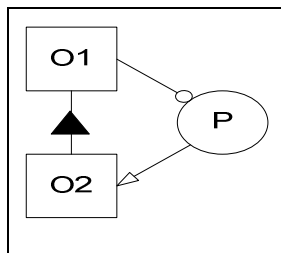
The consumption link between P and O2, that signifies change to O1, contradicts the instrument link between O1 and P, which signifies no change to O1.

**Illegal Construct 2** Part/Attribute Effect and Aggregate/Exhibitor Instrument



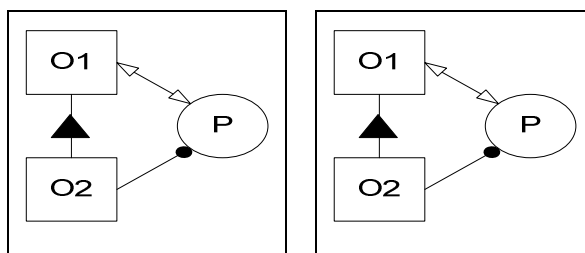
The effect link between P and O2, that signifies change to O1, contradicts the instrument link between O1 and P, which signifies no change to O1.

**Illegal Construct 3** Aggregate Instrument and Part Result



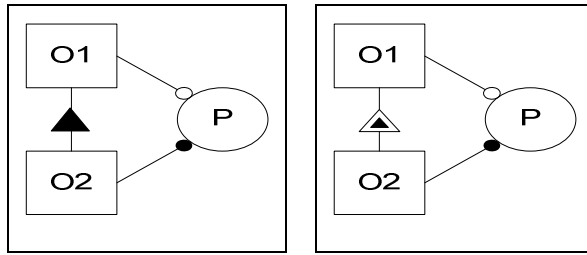
The result link between P and O2, that signifies change to O1, contradicts the instrument link between O1 and P, which signifies no change to O1.

**Illegal Construct 4** Aggregate/Exhibitor Effect and Part/Attribute Agent



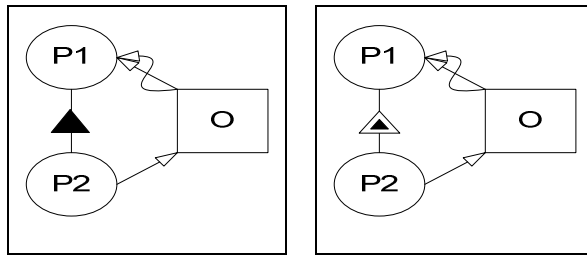
The effect link between O1 and P does not reflect the fact that a part/attribute of O1 is an agent of P.

**Illegal Construct 5** Aggregate/Exhibitor Instrument and Part/Attribute Agent



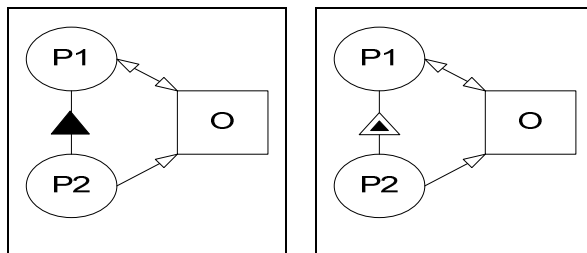
The instrument link between O1 and P does not reflect the fact that a part/attribute of O1 is an agent of P.

**Illegal Construct 6** Aggregate/Exhibitor Consumption and Part/Attribute Result



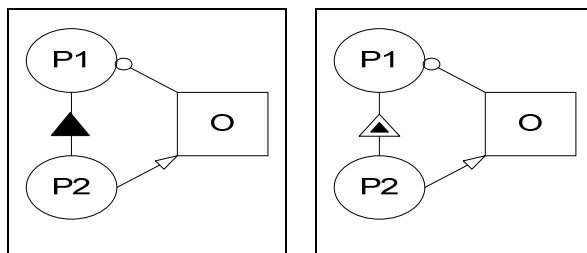
The consumption link between P1 and O contradicts the fact that O is yielded by a part of P1.

**Illegal Construct 7** Aggregate/Exhibitor Effect and Part/Attribute Result



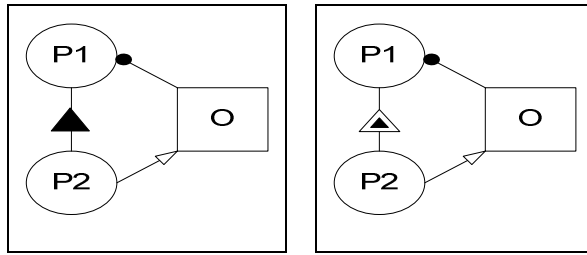
The effect link between P1 and O does not reflect the fact that O is yielded by a part/attribute of P1.

**Illegal Construct 8** Aggregate/Exhibitor Instrument and Part/Attribute Result



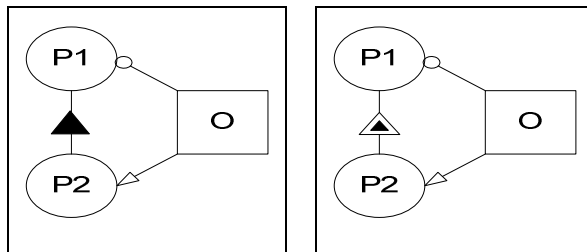
The instrument link between P1 and O does not reflect the fact that O is yielded by a part/attribute of P1.

**Illegal Construct 9** Aggregate/Exhibitor Agent and Part/Attribute Result



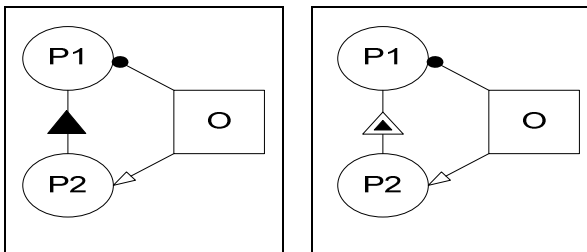
The agent link between P1 and O does not reflect the fact that O is yielded by a part/attribute of P1.

**Illegal Construct 10** Aggregate/Exhibitor Instrument and Part/Attribute Consumption



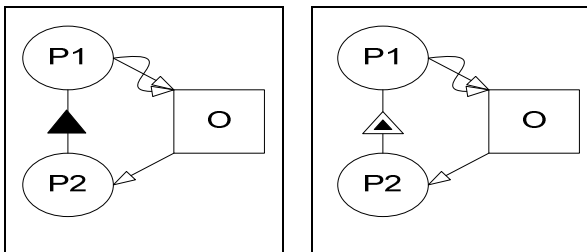
The instrument link between P1 and O does not reflect the fact that O is consumed by a part/attribute of P1.

**Illegal Construct 11** Aggregate/Exhibitor Instrument and Part/Attribute Consumption



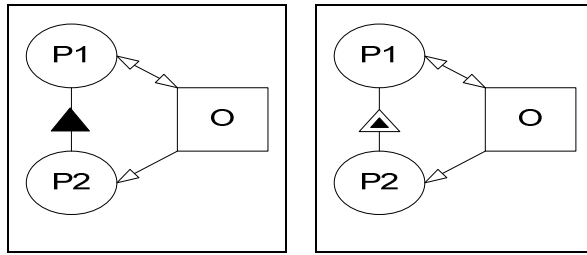
The agent link between P1 and O does not reflect the fact that a part/attribute of P consumes O.

**Illegal Construct 12** Aggregate/Exhibitor Result and Part/Attribute Consumption



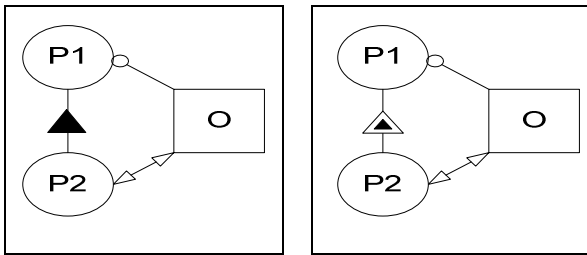
The result link between P1 and O contradicts the fact that O is consumed by a part of P1.

**Illegal Construct 13** Aggregate/Exhibitor Effect and Part/Attribute Consumption



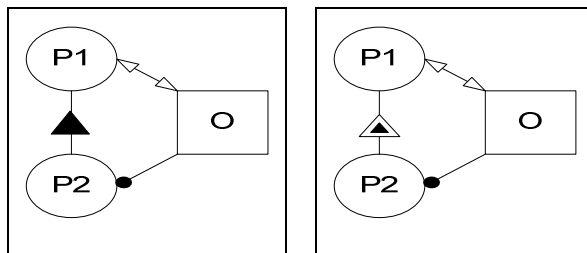
The effect link between P1 and O does not reflect the fact that O is consumed by a part/attribute of P1.

**Illegal Construct 14** Aggregate/Exhibitor Instrument and Part/Attribute Effect



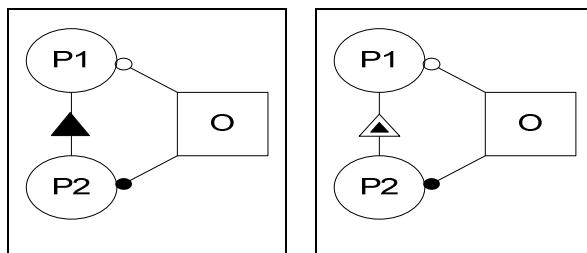
The instrument link between P1 and O does not reflect the fact that O is affected by a part/attribute of P1, specifically P2.

**Illegal Construct 15** Aggregate/Exhibitor Effect and Part/Attribute Agent



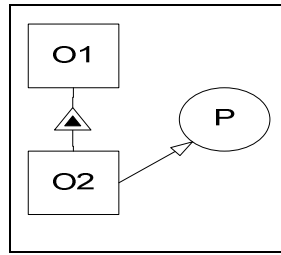
The effect link between P1 and O does not reflect the fact that O is an agent of P2, therefore also an agent of P1.

**Illegal Construct 16** Aggregate/Exhibitor Instrument and Part/Attribute Agent



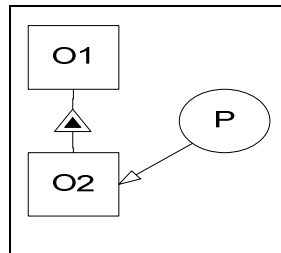
The instrument link between P1 and O does not reflect the fact that O is an agent of P2 and as such also an agent of P1.

**Illegal Construct 17** Attribute Consumption



Attributes are inherent parts of a thing and as such cannot be consumed by any process.

**Illegal Construct 18** Attribute Result



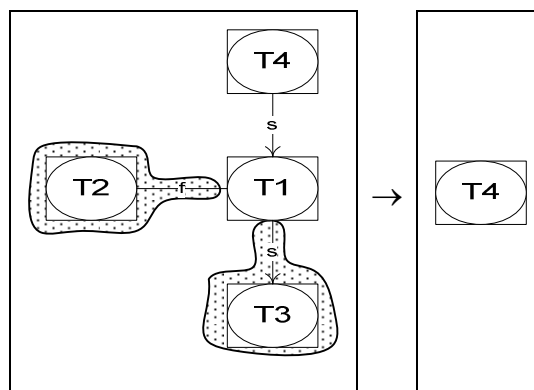
Attributes are inherent parts of a thing and as such cannot be yielded by a process.

**5.3.4 Thing Removal Production**

The final production in the abstraction algorithm tries to remove the processed Thing from the OPD. It checks that the Thing has no procedural link or outgoing structural links, and if so it removes the Thing from the OPD.

**Production 19** Thing Removal

The Thing that is being removed must match with T1



**5.4 Generalization and Classification Abstraction**

Until now the generalization-specialization and classification-instantiation relations have not been considered for the sake of simplicity. To enable complete abstraction of an OPD, these links must also be treated.

In modern programming languages, the generalization/classification relation induces inheritance. Unlike most languages, OPM refers to processes as first class entities that

can exist outside of an object. This information is stored in each thing using a **Thing Type** as defined below.

### 5.4.1 Thing Type and Type Hierarchy

For notation purposes and clarity, each modeled thing in OPM has a type<sup>1</sup>, and the OPM generalization/classification relations create a type hierarchy in the system. Formally, the **Thing Type** is defined as follows:

- Each modeled thing is always of a primitive type. A modeled process named  $\langle process\_name \rangle$  is of type  $\langle process\_name \rangle\_ptype$ . A modeled object named  $\langle object\_name \rangle$  is of type  $\langle object\_name \rangle\_otype$ .
- The primitive type of a thing is denoted as  $T(\langle thing\_name \rangle)$ .

Using the definition above, every thing that is modeled in an OPD has at least one type, its intrinsic primitive type, which is shown in the OPD model by the combination of the shape of the thing (rectangle or ellipse) and the name of the thing

Using the OPM generalization-specialization relation, we can define that one thing specializes another thing, taking its features (i.e., attributes and operations), relations and states<sup>2</sup>. This means that the specialized thing becomes of the type of the general entity. Formally, when two things are connected in a generalization-specialization relation, the specialized thing (the target of the relation) acquires the type of the general thing (the source of the relation).

For example, suppose P1 and P2 are two processes. From the definition above, process P1 is of type  $P1\_ptype$  and P2 is of  $P2\_ptype$ . Furthermore, suppose that P1 and P2 are connected with a generalization-specialization link where P1 is the source of the link and P2 is the target of the link. Then P2 is also of type  $P1\_ptype$ .

We have defined that every thing in OPM has a type, and when a complex OPM static hierarchy is created, a thing can be of many different types. But how do we know of what types a thing is? For this purpose we define the **Type Closure** of a thing. The type closure of a thing, denoted by  $TC(\langle thing\_name \rangle)$  is a group of types, such that

---

<sup>1</sup> The word type rather than class has been selected since class is used in most languages to define objects, and this might confuse the reader.

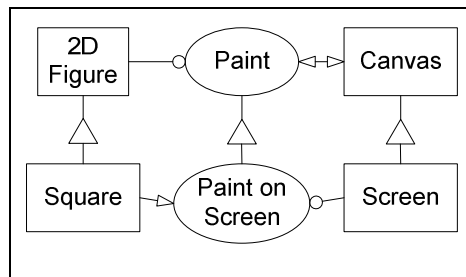
<sup>2</sup> The Generalization-Specialization relation in OPM is usually called inheritance in modern object oriented languages.

- $$TC(\langle thing1 \rangle) = \left[ \bigcup_{\langle thing1 \rangle \triangleright \langle thing2 \rangle} TC(\langle thing2 \rangle) \right] \cup T(\langle thing1 \rangle),$$
 where  $\langle thing1 \rangle \triangleright \langle thing2 \rangle$  denotes that  $thing1$  is a  $thing2$ , as modeled in the OPD. Furthermore, if  $thing1$  is an instance of  $thing2$ , then  $TC(\langle thing1 \rangle) = TC(\langle thing2 \rangle) \cup T(\langle thing1 \rangle)$ .

### 5.4.2 Syntactic Validation of Generalization and Classification

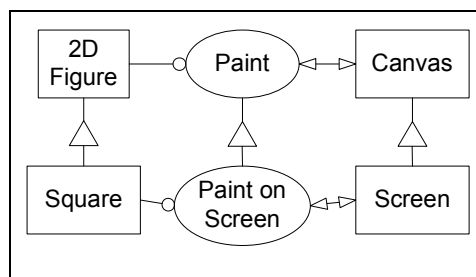
The generalization-classification relation can link two objects or two processes. When the relation is between two objects, there is no syntactic check that can be done, since the structure of the things in the OPD is completely up to the modeler to decide.

When the relation links two processes, the abstraction algorithm must validate that the "signature" or "API"<sup>3</sup> of the source of the relation is maintained in the target of the relation. The API of a process consists of all incoming and outgoing procedural links, including the object types that are at the ends of these links. For example, Figure 41 shows an invalid OPD.



**Figure 41. Invalid signature example**

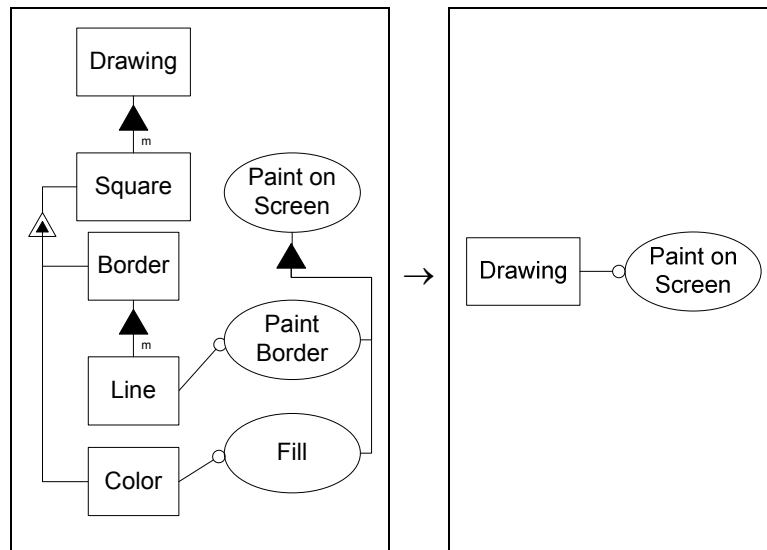
Process **Paint** requires **2D Figure** and affects **Canvas**. **Paint on Screen** is a **Paint**, therefore it must conform to the same signature as **Paint**, meaning that it must require an object of type **2D Figure** and must affect an object of type **Canvas**. Since this is not the case, this OPD is invalid. A valid version of this OPD is shown in Figure 42.



<sup>3</sup> API – An application programming interface (API) is a source code interface that an operating system or library provides to support requests for services to be made of it by computer programs

**Figure 42. Valid signature abstraction**

The signature of the process must be maintained, but may be expanded by creating new links in addition to the links that already exist in the original signature. Furthermore, the signature of a process is defined by the procedural links that are connected to the process at modeling time before abstraction. This detail is important since the signature of a process can change during the abstraction process in unpredictable ways.

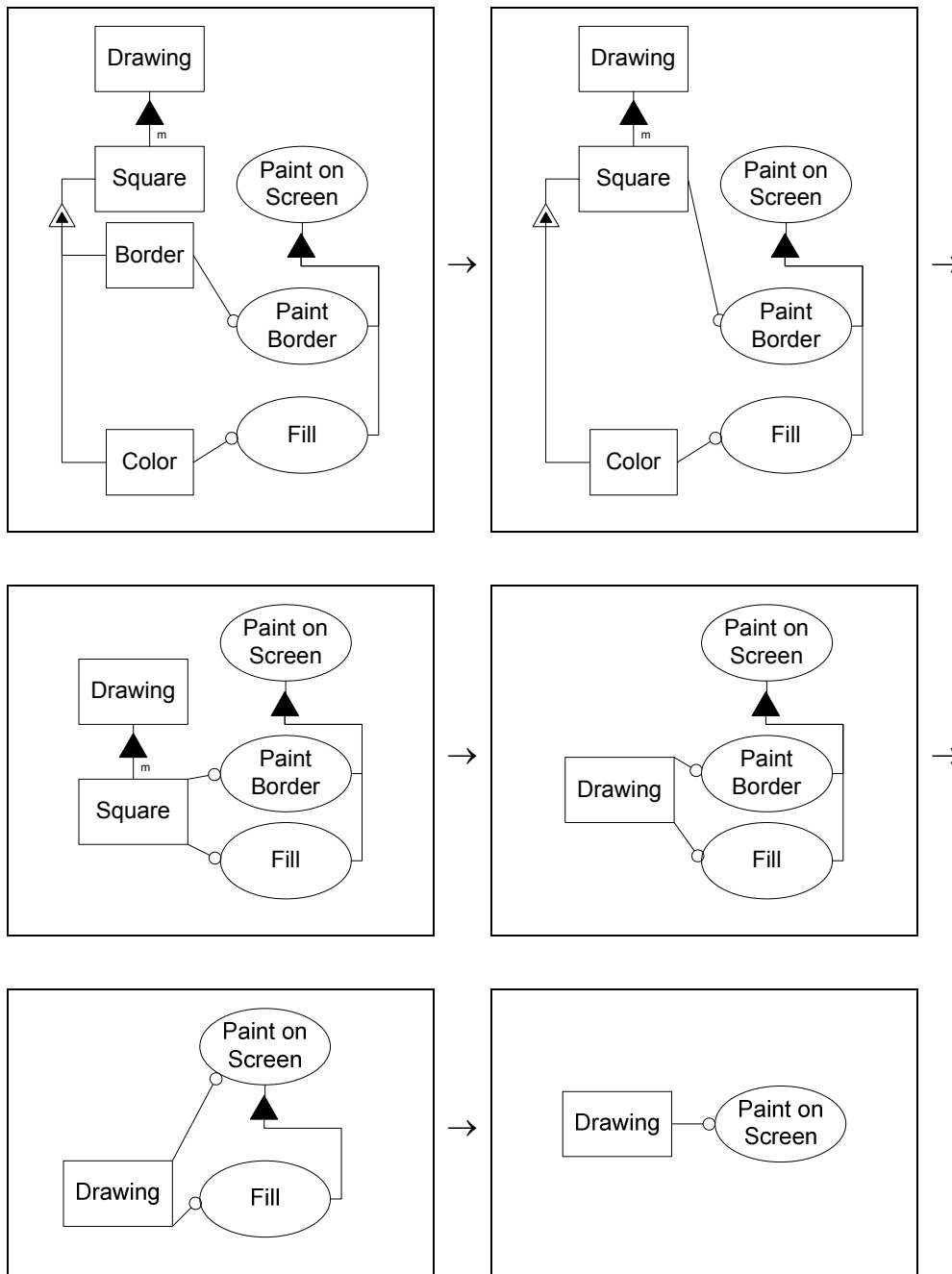


**Figure 43. Example OPD before and after abstraction**

Figure 43 demonstrates this. The original OPD on the left hand side yields the final abstraction after applying all possible graph grammar productions. Interestingly, for abstraction purposes this is correct – There is a process called **Paint on Screen** which requires **Drawing**. On the other side, **Paint on Screen** is a **Paint**, and this means that it requires a **2D Figure**, and **Drawing** is never defined to be a **2D Figure**. Furthermore, there is no guarantee that at any point in the abstraction process **Square** will be linked to **Paint On Screen**, as can be seen in the run of the abstraction algorithm on our example in Figure 44<sup>4</sup>.

---

<sup>4</sup> This example run is specially tailored for the example, and although many other possible runs may occur, this one is certainly possible.



**Figure 44. Abstraction process example execution**

Due to this reason, the signature of a process must be maintained at modeling time, even at the cost of extra redundant links in the OPD. This is done using the Signature Consistency Validation algorithm described below.

**Algorithm 3** Signature Consistency Validation

Let P1 and P2 be two processes connected in a generalization or classification link, where P1 is the source and P2 is the target of the link. Let  $IL_{P1}$  and  $OL_{P1}$  be the sets of all incoming and outgoing links to P1 respectively, and  $IL_{P2}$  and  $OL_{P1}$  the set of

all incoming and outgoing links of P2 respectively. Denote by  $s(l)$  the source and  $t(l)$  the target of link  $l$ .

- For each  $l_{p1} \in IL_{p1}$ 
  - Find  $l_{p2} \in IL_{p2}$  such that  $T(s(l_{p1})) \in TC(s(l_{p2}))$ . If no such link exists, stop and return error. Otherwise, remove  $l_{p2}$  from  $IL_{p2}$  ( $IL_{p2} = IL_{p2} \setminus \{l_{p2}\}$ ).
- For each  $l_{p1} \in OL_{p1}$ 
  - Find  $l_{p2} \in OL_{p2}$  such that  $T(t(l_{p1})) \in TC(t(l_{p2}))$ . If no such link exists, stop and return error. Otherwise, remove  $l_{p2}$  from  $OL_{p2}$  ( $OL_{p2} = OL_{p2} \setminus \{l_{p2}\}$ ).

After the algorithm above is executed, all the links that exist in P1 have been validated, checking that they also exist in P2.

### 5.5 Complete OPD Abstraction Algorithm

Extending the algorithm defined in section 5.2, the following algorithm completes the abstraction of the OPD to its minimal abstraction.

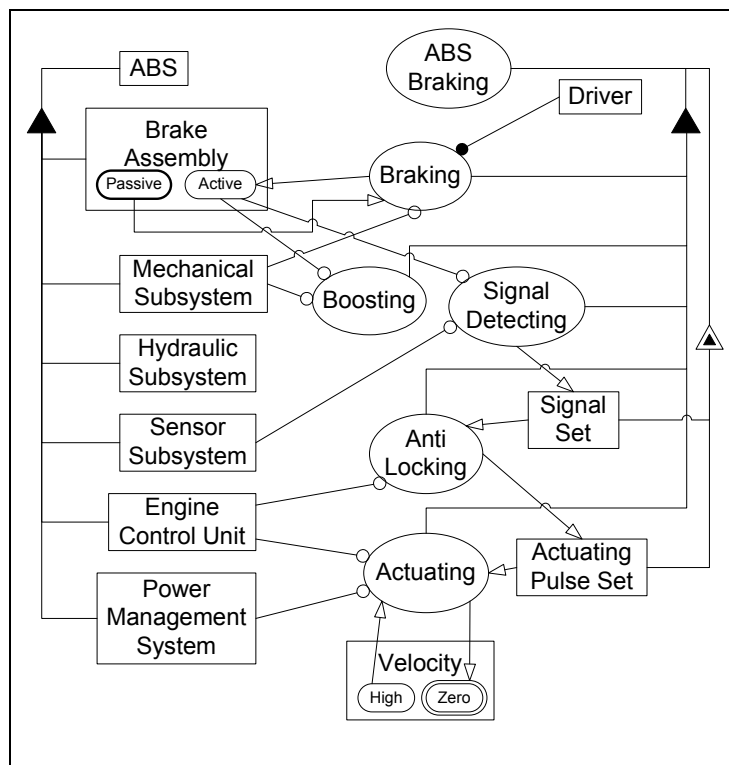
- Input: OPD
- Algorithm:
  1. Calculate *Type* and *Type Closure* of all things in the OPD.
  2. Validate all Process signatures by applying the Signature Consistency Validation algorithm. If validation failed, stop and return failure on signature validation.
  3. While OPD contains things that have not been processed:
    - 3.1. Of all things in the current OPD select **thing** with  $\max(\text{height}(\text{thing}))$  and no outgoing structural links.
    - 3.2. Transform all *Temporary Links* that start at **thing** to *Regular Links*.
    - 3.3. Apply *State Change Abstraction* production to **thing** if applicable, as many times as possible.
    - 3.4. Apply *State-Specified Link Abstraction* production to **thing** if applicable, as many times as possible.
    - 3.5. Apply *Procedural Abstraction* productions to **thing** if applicable, as many times as possible.
    - 3.6. Check *Illegal Constructs* on **thing**. If *illegal constructs* exist, stop and return failure on **thing**.

- 3.7. Apply *Thing Removal* production to **Thing** if applicable. If the production is not applicable, mark **Thing** as processed.
4. Transform all *temporary links* in the OPD to *regular links*.
5. End.

**Example 3** ABS Braking OPD Abstraction

In this example we try to abstract SD1 of the ABS Ford system model example provided by OPCAT. The OPD (flattened to remove in-zooming) is shown Figure 45.

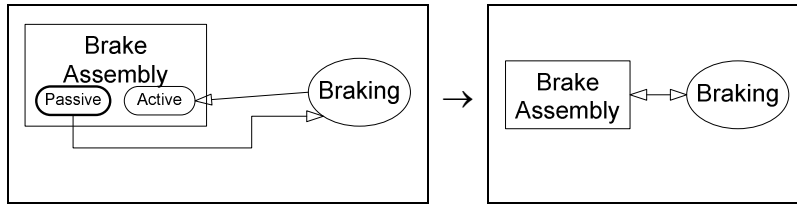
ABS Braking OPD



**Figure 45. ABS Braking OPD**

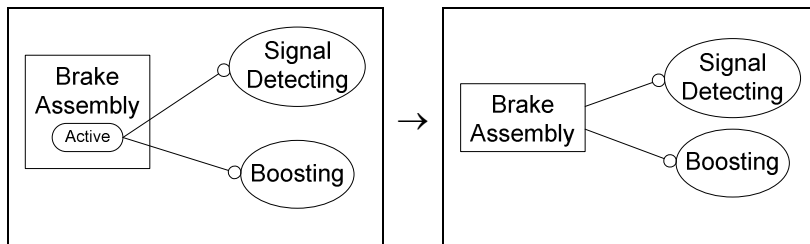
In the example OPD there are things of with heights 1 and 2. Since there are no generalization or classification links in the diagram, the first step is to go over all the things of height 2 and apply the abstraction steps. We first process all the objects and then all the processes, starting with object **Brake Assembly**.

The first step is to transform all temporary links. Since there are none, this step is finished. The next step is to apply *State Change Abstraction*. This step can be applied to **Brake Assembly** using the link that starts at state **Passive** and ends at **Braking**, and the link that starts at state **Braking** and ends at **Active**. The result of this step is shown in Figure 46. Since most of the diagram remains the same, only the affected part is shown.



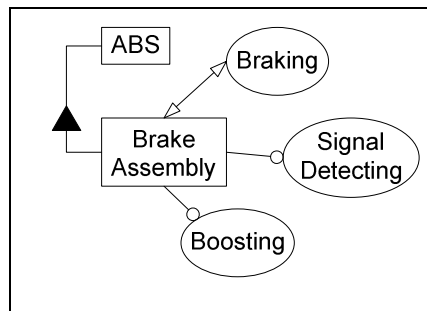
**Figure 46. State change abstraction on Brake Assembly**

The next step is to apply *State-Specified Link abstraction*. There are two links that start at a state of **Brake Assembly**, one that starts at state **Active** and ends at **Boosting** and the other that starts at **Active** and ends at **Signal Detecting**. The result of this step (one again removing unnecessary parts of the diagram) is shown in Figure 47.



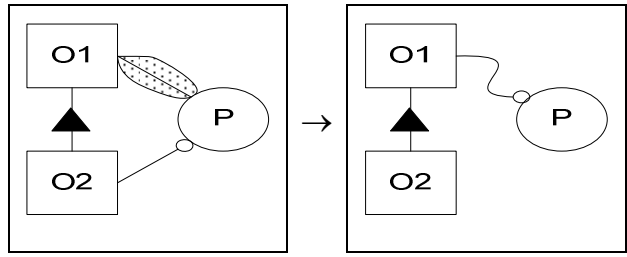
**Figure 47. State-Specified Link abstraction on Brake Assembly**

The next step is *Procedural Abstraction*. In this step, the procedural links that connect **Brake Assembly** to all other things in the diagram are "transferred" to its structural parent, which is **ABS**. The current status of the diagram (removing the irrelevant elements) is shown in Figure 48.



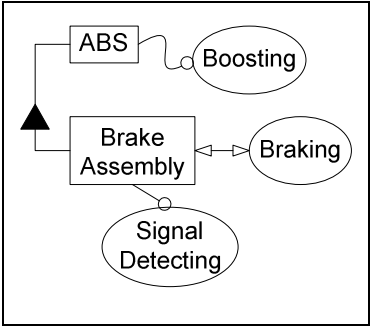
**Figure 48. Example diagram after initial abstraction of Brake Assembly**

The first link that will be abstracted is the link to **Boosting**. Using the productions defined in section 5.3.2, the matching production for this case is Production 9.5.1, shown in Figure 49.



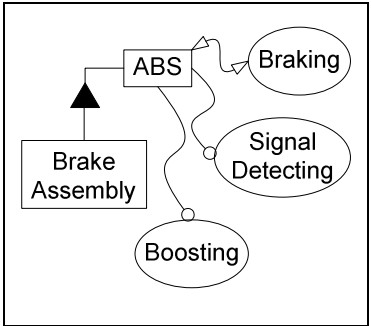
**Figure 49. Promotion of Part Instrument to Aggregate Instrument Production**

The production is applied to the example diagram resulting in the diagram shown in Figure 50.



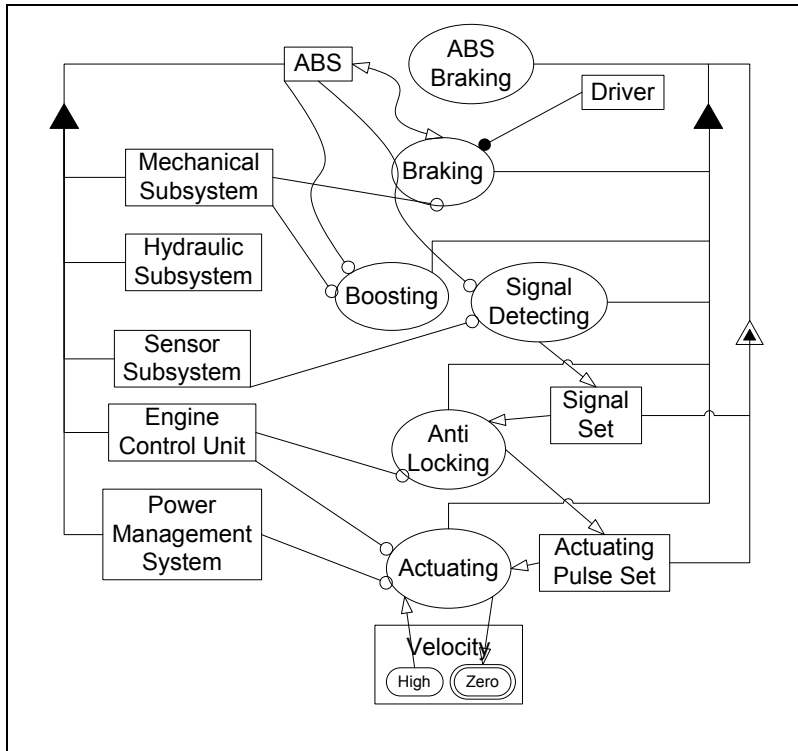
**Figure 50. Example diagram after Brake Assembly instrument link abstraction**

The links from **Brake Assembly** to **Braking** and **Signal Detecting** are abstracted using similar productions, creating the diagram shown in Figure 51.



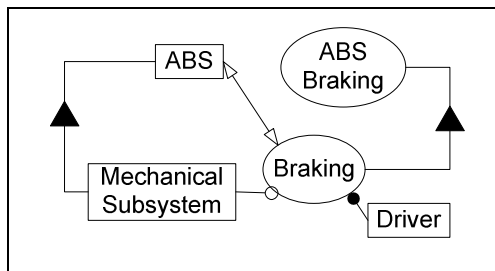
**Figure 51. Example diagram after all Brake Assembly links abstracted**

No *illegal constructs* were detected on **Brake Assembly**, so the next step is *Thing Removal*. The result of the first round of the algorithm is shown in Figure 52.



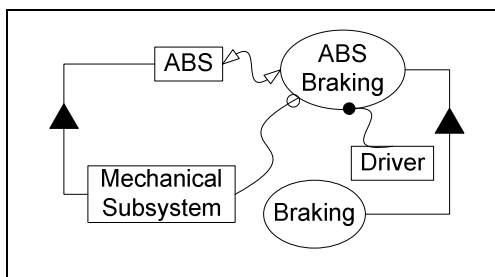
**Figure 52. Example diagram after first round of the abstraction algorithm**

A process is abstracted using the same steps used to abstract an object. For example, process **Braking** will be the one abstracted next. After transforming the temporary links starting at it, the working diagram (removing unnecessary things) is shown in Figure 53.



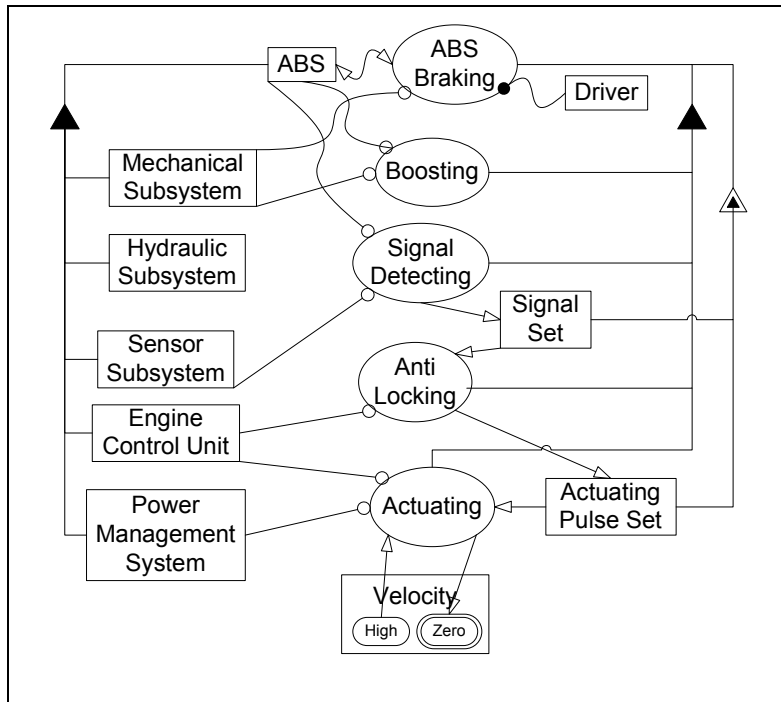
**Figure 53. Example diagram before **Braking** process abstraction**

The steps used to abstract a process are in general fewer than those used to abstract an object since a process does not contain states. Hence the first step is *Procedural Abstraction*. The result of applying this step is shown in Figure 54.



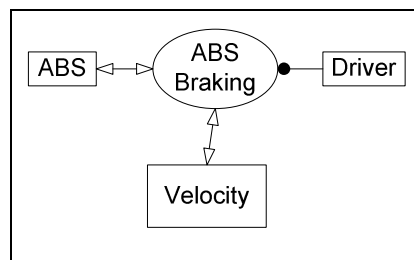
**Figure 54. Example diagram after procedural abstraction of Braking process**

After this, **Braking** can be removed from the full diagram, yielding the diagram shown in Figure 55.



**Figure 55. Example diagram after abstraction of Braking process**

The abstraction process continues in the same way until there are no more things that can be abstracted. Then, all the temporal links are transformed to regular links. The final diagram is shown in Figure 56.



**Figure 56. Final abstraction of the example diagram**

## **6. Coverage**

This work has treated the abstraction of many of the existing constructs in OPM. Nonetheless, there are some that have only been treated partially and some that have not been treated at all. This section reviews what has been treated, what not, and why.

### ***6.1 Structural Relations***

All types of structural relations have been treated in this work: aggregation-participation, exhibition-characterization, generalization-specialization, classification-instantiation and tagged relations. These relations are fully handled when their source and destination is a thing (object or process). Structural relations whose source or destination is a state have not been treated since they add a large number of special cases that must be treated specifically. Most of these relations can be abstracted to start at the containing object and at this stage the abstraction algorithm may be applied.

There is no validation of structural loops. There are cases where loops are valid and there are others where loops are invalid. The current work allows any type of loop to exist in the diagram.

### ***6.2 Procedural Relations***

The basic procedural links have been treated in this work, including consumption, effect, result, agent, instrument and invocation links. These links are abstracted by adding them to the aggregating or exhibiting thing and using temporary links to find the final abstracted links between two things.

Event, conditions and exception links are not treated at all. Event and condition links can be abstracted to instrument (instrument event link and condition link) or consumption (consumption event link) links with minimal reduction of the semantics of the OPD. Exception links are a separate category that must be treated specifically in a possible extension of this research.

### ***6.3 States***

States have been treated in this work largely as semantic parts of object that can be easily abstracted and removed. The use in OPM of states as values was not treated.

#### ***6.4 Abstraction: In-Zooming and Unfolding***

In-zooming and unfolding are not treated in this work. Theoretically (as stated above) a system can be specified in a single flat and complex OPD, as the use of abstraction is only for human understandability purposes. Nonetheless, since the abstraction facilities OPM are one of the things that make it such a strong modeling language, an additional discussion in section 7 is dedicated to these facilities and how the verification can be done on them.

## 7. From a single OPD to a Complete OPM System Validation

We have shown how to validate an OPD in its creation phase and when it is finished (or at any time the modeler deems necessary). One of the most important traits of OPM is its ability to abstract and refine information with its out-zooming and in-zooming and with its folding and unfolding operations. In this section we refer to this capability noting two things:

- 1) The in-zooming and unfolding operations create new OPDs where the OPD creation rules apply. The important step here is to "save" the links that exist between things in the parent OPD (The OPD where the thing is in-zoomed or unfolded – note that there may be many of these) and the child OPD.
- 2) A System Model that consists of many OPDs can be validated by validating each OPD recursively.

This will be further expanded below. However note that this is a preliminary work and not a full investigation of the problem, therefore the algorithms and definitions may not be completely formal or may contain limitations that are not existent in OPM.

### 7.1 *Preliminary Definitions*

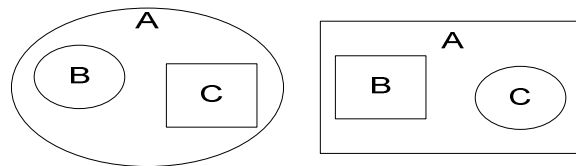
#### 7.1.1 In-Zooming and Unfolding

The in-zooming operation and the unfolding operation, when applied to a Thing in an OPD, create a new OPD that is used by the modeler to show more information about the thing that is in-zoomed or unfolded. This information is not shown in the OPD where the in-zooming/unfolding was applied because it would probably create a lot of clutter; therefore a new OPD is created to show this information.

There is a direct connection between the in-zoomed/unfolded entity in all the OPDs where it is seen as out-zoomed/folded and the in-zoomed/unfolded OPD where this thing is the central one. In OPM, an in-zoomed/unfolded thing is denoted by a bold border on its symbol (in OPCAT [6] this operation also adds color to the in-zoomed/unfolded thing in some cases).

An unfolding-refined OPD is a simple OPD where the unfolded thing occurs at the top of the OPD. Furthermore, the thing that is unfolded may not be removed from the OPD.

An in-zoom-refined OPD contains the in-zoomed thing which is enlarged at the center of the in-zoom-refined OPD. This allows for the addition of things *inside* the in-zoomed thing, as shown in Figure 57.



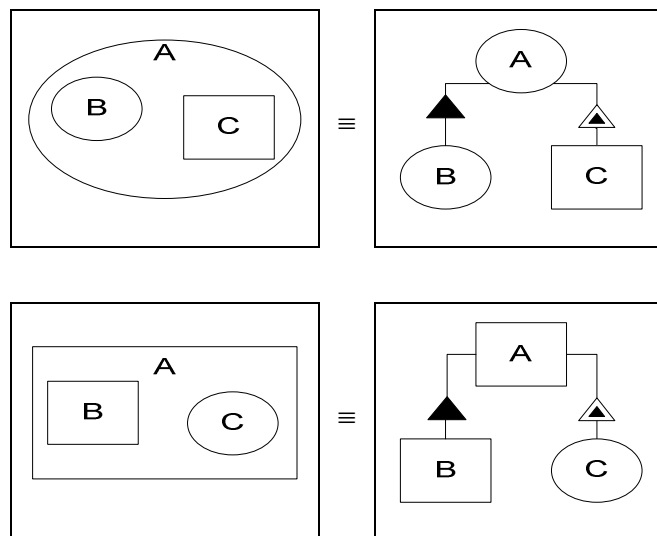
**Figure 57. In-Zoom Refinement**

In Figure 57 there are two examples of in-zooming. On the left hand side, process *A* is in-zoomed, and is shown to contain process *B* and object *C*. On the right hand side, object *A* is in-zoomed, and it contains object *B* and process *C*.

The things that are inside an in-zoomed thing are related to it in the following manner.

- 1) In-zoomed Process:
  - a) Aggregates the processes that are inside it, and
  - b) Exhibits the objects that are inside it.
- 2) In-zoomed Objects:
  - a) Aggregates the objects that are inside it, and
  - b) Exhibits the processes that are inside it.

These relations are presented graphically in Figure 58.



**Figure 58. In-Zoom Refinement Unfolding Transformation**

The links that connect an entity inside an in-zoomed thing are the same links that connect it outside it. The in-zooming of a thing is used for clarity, and does not change the meaning of the constructs in OPM. However, in an in-zoomed process the timeline is from top to bottom and this is used for modeling synchronicity among processes. Unfolding of process to subprocesses models asynchronous systems.

### 7.1.2 Procedural Link Abstraction between a Pair of Things

The procedural link that connects two things in an in-zoom-refined OPD may be different from the link that connects the two things in the out-zoomed OPD. This can occur when the in-zoomed OPD shows only part of the functionality of the out-zoomed OPD. For the links to be valid between the two related OPDs, the link that exists in the out-zoomed OPD must abstract the link in the in-zoomed OPD. The abstraction order of procedural links is shown in Figure 59. Because the consumption and result link have the same symbol in OPM, the link that goes from left to right is the consumption link and the link that goes from right to left is the result link (to remember this, suppose always that an object occurs at the left side of the link and a process occurs at the right side of the link).

In-zoomed diagram \ Out-zoomed diagram	—●	—○	—▷	◁—	◁▷
—●	V	V	V	V	V
—○	X	V	X	X	X
—▷	V	V	V	X	X
◁—	V	V	X	V	X
◁▷	X	V	V	V	V

**Figure 59. The link precedence matrix**

The table in Figure 59 is used as follows:

1. Find the link in the in-zoomed diagram in the first (topmost) row of the table.
2. Find the link in the out-zoomed diagram in the first (leftmost) column of the table.
3. If the intersection of the selected column and the selected row contains a **V**, the link in the out-zoomed diagram abstracts the link in the in-zoomed diagram.
4. If the intersection of the selected column and the selected row contains an **X**, the link in the out-zoomed diagram does not abstract the link in the in-zoomed diagram

### 7.1.3 Matching Thing between Parent and Child OPD

After an OPD has been abstracted to its final abstraction, if it is not the topmost OPD of the system it must be out-zoomed/folded into the OPD from which it was created. This works only for in-zoomed diagrams because in them the parent diagram can be found using the OPD hierarchy, which cannot be done in unfolded diagrams since they always occur in the topmost level of the OPM hierarchy.

To match two OPD it is assumed that there is only one thing of the same kind (object/process) and the same name in the OPDs. In other words, if the parent OPM contains an object named `Card` and the child OPD contains an object named `Card`, they both refer to the same object. This requirement must also be met for processes.

It is assumed that all the things in the in-zoomed OPD have no structural relations because it is in its minimal abstraction state. If this is not the case this can be easily fixed by moving the relation to the unfolded OPD.

The match is done as follows. Let  $SDa$  be a diagram and  $SDa.b$  the in-zoomed diagram of a thing that exists in  $SDa$ . Note that  $SDa.b$  must be in the minimal abstraction as defined in section 5.5:

1. Verify that all things in  $SDa.b$  of the same kind have different names. If there are two things of the same kind with the same name, stop and return error on thing.
2. For each thing in  $SDa.b$ 
  - 2.1. Find a thing in  $SDa$  that is of the same kind and has the same name. If there is no such thing, stop and return error on thing.
  - 2.2. Mark the matched thing in  $SDa$  to the matched thing in  $SDa.b$ .
3. For each procedural link that connects between two things in  $SDa.b$ 
  - 3.1. If there is no procedural link between the two things in  $SDa$ , add the procedural link that exists in  $SDa.b$ .
  - 3.2. If there is a procedural link between the two things in  $SDa$ , verify that this link is the same or an abstraction of the link that exists in  $SDa.b$ .

## 7.2 *Full System Model Validation*

The validation of a full system model can be done in a number of ways:

- 1) Validate each OPD using the validation algorithm shown in section 5.5 recursively, by validating first the OPDs of higher level and from there going down to the OPDs of lower level.

- 2) Create one OPD that contains a union of all of the OPDs that exist in the system model. This can be done recursively starting from SD, the highest level OPDs and including them in the OPDs from which SD is in-zoomed or unfolded.

Option 1 is simpler and more straightforward, therefore this is the option that will be implemented. The algorithm to do this is as follows:

1. While there are OPDs in the OPD hierarchy other than SD
  - 1.1. Select the OPD with the highest level. If there is more than one, select one arbitrarily.
  - 1.2. Transform the in-zoomed OPD as described in section 7.1.1.
  - 1.3. Apply the OPD abstraction algorithm defined in section 5.5.
  - 1.4. Match the links in the in-zoomed OPD to the links in the un-zoomed OPD as described in section 7.1.3.
  - 1.5. If all of the previous steps completed successfully, delete the in-zoomed OPD.

## 8. Conclusions

In this research we developed a formal definition of the syntax of an OPD and a method for the creation and verification of OPDs. This formalization provides OPM with a solid foundation in the software engineering field, as the syntactic and some of the semantic correctness of models and diagrams can be verified. This formal verification is critical as we wish to create robust and verifiable systems.

A formal and exact definition of the syntax and the semantics of OPM opens the way to a large number of possibilities. These include validation and automatic testing of systems at design time. These two items can be of great help to systems and software engineering where nowadays testing occurs mostly at the end of the software coding or during production and errors in these stages are very costly and difficult to correct.

Another possibility that becomes open is system lifecycle management. As a system is changed, these changes can be done in the system model and can be compared semantically to the original model to detect what are the parts of the model affected by the change. This can greatly reduce the amount of testing needed when new versions of a system are produced. Another use can occur when a part of a system needs to be changed for some reason (i.e., a part that was memory expensive in the system must be changed). The semantic verification of the model can check whether the changed part of the model still produces the same functionality that the original part. This verification in conjunction with automatic model testing can give a high degree of confidence that the changes made in the system will not affect its functionality.

One more use that comes to mind is automatic optimization of system models. Having a formal definition of the syntax and semantics of OPM enable the creation of automatic programs that can read system diagrams and optimize them for memory use or speed, to name a few. Many system models are redundant and have many recurring parts that exist in the system because the system engineers are human beings, and no matter how intelligent they are, as the system gets larger they cannot have it all in their minds at one time and therefore duplication of existing functionality may occur. Since clarity and simplicity are one of the main goals of system design, at times this comes at the expense of performance. Therefore an optimization phase after system modeling can be very effective.

## 9. References

- [1] Balbo, G.; Desel, J.; Jensen, K.; Resig, W.; Rozenberg, G. & Silva, M. (2000), Petri Nets 2000: Introductory Tutorial, *in* '21st International Conference on Application and Theory of Petri Nets'.
- [2] Breu, R.; Hinkel, U.; Hofmann, C.; Klein, C.; Paech, B.; Rumpe, B. & Thurner, V. (1997), Towards a Formalization of the Unified Modeling Language, *in* 'ECOOP', pp. 344-366.
- [3] Bruel, J. & France, R. B. (1998), Transforming UML models to Formal Specifications, *in* Pierre-Alain Muller & Jean Bézivin, ed., 'Proc. International Conference on the Unified Modelling Language (UML): Beyond the Notation', Springer-Verlag.
- [4] Corradini, A.; Ehrig, H.; Heckel, R.; Korff, M.; Lowe, M.; Ribeiro, L. & Wagner, A. (1997), Algebraic Approaches to Graph Transformation - Part I: Single Pushout Approach and Comparison with Double Pushout Approach, *in* G. Rozenberg, ed., 'Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations', World Scientific, pp. 247-312.
- [5] Dijkstra, E. W. (1972), *Chapter I: Notes on structured programming*, Academic Press Ltd., London, UK, pp. 1-82.
- [6] Dori, D. (1996), 'Object-process analysis of computer integrated manufacturing documentation and inspection functions', *International Journal of Computer Integrated Manufacturing* **9**(5), 339-353.
- [7] Dori, D. & Crawley, E. F. (1999), *Object-Process Methodology: A Holistic Systems Paradigm*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- [8] Dori, D. & Sturm, A. (1998), OPCAT - Object-Process Case Tool: an Integrated System Engineering Environment (ISEE), *in* 'ECOOP Workshops', pp. 555-556.
- [9] Ehrig, H.; Ehrig, K.; Habel, A. & Pennemann, K. (2004), Constraints and Application Conditions: From Graphs to High-Level Structures, *in* 'ICGT 2004', pp. 287-303.
- [10] Ehrig, H.; Heckel, R.; Korff, M.; Luwe, M.; Ribeiro, L.; Wagner, A. & Corradini, A. (1997), Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach, *in* 'Handbook of Graph Grammars and Computing by Graph Transformation. Vol.

- I: Foundations', World Scientific, pp. 247-312 .
- [11] Engels, G.; Heckel, R. & Sauer, S. (2000), UML - A Universal Modeling Language?, *in* 'ICATPN 2000', pp. 24-38.
- [12] Evans, A.; France, R. B.; Lano, K. & Rumpe, B. (1999), The UML as a Formal Modeling Notation, *in* 'UML '98: Selected papers from the First International Workshop on The Unified Modeling Language UML', Springer-Verlag, London, UK, pp. 336-348.
- [13] France, R. B.; Ghosh, S.; Dinh-Trong, T. & Solberg, A. (2006), 'Model-Driven Development Using UML 2.0: Promises and Pitfalls', *Computer* **39**(2), 59.
- [14] Gogolla, M. & Parisi-Presicce, F. (1998), State Diagrams in UML: A Formal Semantics using Graph Transformations, *in* Manfred Broy; Derek Coleman; Tom S. E. Maibaum & Bernhard Rumpe, ed., 'Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques', Technische Universitat München, TUM-I9803, .
- [15] Gogolla, M.; Ziemann, P. & Kuske, S. (2003), 'Towards an Integrated Graph Based Semantics for UML', *Electronic Notes in Theoretical Computer Science* **72**(3).
- [16] Habel, A.; Heckel, R. & Taentzer, G. (1996), 'Graph grammars with negative application conditions', *Fundam. Inf.* **26**(3-4), 287-313.
- [17] Harel, D. (1987), 'Statecharts: A Visual Formulation for Complex Systems', *Science Computerts Programming* **8**(3), 231-274.
- [18] Harel, D. & Kugler, H. (2004), The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML), *in* 'Integration of Software Specification Techniques for Application in Engineering', Springer-Verlag, pp. 325-354.
- [19] Heckel, R. (1995), Embedding of conditional graph transformations', *in* Valiente Feruglio G. & Rosello Lompart F., ed., 'Proceedings Colloquium on Graph Transformation and its Application in Computer Science', Technical Report B-19, Universitat de les Illes Balears.
- [20] Jürjens, J. (2002), A UML statecharts semantics with message-passing, *in* 'SAC', pp. 1009-1013.
- [21] Kastenbergh, H.; Kleppe, A. & Rensink, A. (2006), 'Engineering Object-Oriented Semantics using Graph Transformations'(TR-CTIT-06-12), Technical report, Department of Computer Science, University of Twente.
- [22] Kobryn, C. (2004), 'UML 3.0 and the future of modeling', *Software and Systems*

*Modeling* **3**(1), 4-8.

- [23] Kong, J.; Zhang, K.; Dong, J. & Song, G. (2003), A Graph Grammar Approach to Software Architecture Verification and Transformation, *in* 'COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications', IEEE Computer Society, Washington, DC, USA, pp. 492.
- [24] Kuske, S. (2001), A Formal Semantics of UML State Machines Based on Structured Graph Transformation, *in* 'Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools', Springer-Verlag, London, UK, pp. 241-256.
- [25] Kuske, S.; Gogolla, M.; Kollmann, R. & Kreowski, H. (2002), An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation, *in* 'IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods', Springer-Verlag, London, UK, pp. 11-28.
- [26] Mayer, R. E. (2001), *Multimedia Learning*, Cambridge University Press, New York, NY, USA.
- [27] Miller, J. & Mukerji, J. (2003), 'MDA Guide Version 1.0.1', Technical report, Object Management Group (OMG).
- [28] Mwaluseke, G. W. & Bowen, J. P. (2001), 'UML Formalisation Literature Survey'.
- [29] Nestor, A. O., 'Modeling of large and complex applications with UML'.
- [30] Object Management Group (2007), 'SysML Specification', at <http://www.sysml.org/>.
- [31] Object Management Group (2003), 'Unified Modeling Language (UML) 2.0 Infrastructure Specification', at <http://www.uml.org/>.
- [32] Object Management Group (2003), 'Unified Modeling Language (UML) 2.0 Superstructure Specification', at <http://www.uml.org/>.
- [33] Peleg, M. & Dori, D. (2000), 'The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods', *IEEE Trans. Softw. Eng.* **26**(8), 742-759.
- [34] Peleg, M. & Dori, D. (1999), 'Extending the Object-Process Methodology to Handle Real-Time Systems', *Journal of Object-Oriented Programming* **11**(8), 53-58.
- [35] Reinhartz-Berger, I. & Dori, D. (2005), A Reflective Metamodel of Object-Process Methodology: The System Modeling Building Blocks, *in* Peter Green &

- Michael Rosemann, ed., 'Business Systems Analysis with Ontologies', Idea Group.
- [36] Reinhartz-Berger, I.; Dori, D. & Katz, S. (2002), 'OPM/Web – Object-Process Methodology for Developing Web Applications', *Ann. Softw. Eng.* **13**(1-4), 141-161.
- [37] Schmidt, D. C. (2006), 'Guest Editor's Introduction: Model-Driven Engineering', *IEEE Computer* **39**(2), 25-31.
- [38] Schürr, A.; Winter, A. & Zündorf, A. (1995), Graph Grammar Engineering with PROGRESS, in 'Software Engineering – ESEC '95', Springer Berlin / Heidelberg, pp. 219-234
- [39] Snook, C. & Butler, M. (2006), 'UML-B: Formal modeling and design aided by UML', *ACM Trans. Softw. Eng. Methodol.* **15**(1), 92-122.
- [40] Soffer, P.; Golany, B. & Dori, D. (2003), 'ERP modeling: a comprehensive approach', *Inf. Syst.* **28**(6), 673-690.
- [41] Soffer, P.; Golany, B.; Dori, D. & Wand, Y. (2001), 'Modelling Off-the-Shelf Information Systems Requirements: An Ontological Approach', *Requir. Eng.* **6**(3), 183-199.
- [42] Spivey, J. M. (1989), *The Z notation: A Reference Manual*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [43] Stürle, H. & Hausmann, J. H. (2005), Towards a Formal Semantics of UML 2.0 Activities, in 'Software Engineering', pp. 117-128.
- [44] Tchertchago, A. (2002), 'Formal Semantics for a UML Fragment Using UML/OCL Metamodeling'.
- [45] Thomas, D. (2004), 'MDA: Revenge of the Modelers or UML Utopia?', *IEEE Software* **21**(3), 15-17.
- [46] Vanderperren, Y. & Dehaene, W. (2005), UML 2 and SysML: An Approach to Deal with Complexity in SoC/NoC Design, in 'DATE', pp. 716-717.
- [47] Wand, Y. & Weber, R. (1993), 'On the Ontological Expressiveness of Information Systems Analysis and Design Grammars', *Journal of Information Systems*, 217-237.
- [48] 'Wikipedia - The Free Encyclopedia', at <http://wikipedia.org/>.
- [49] Ziemann, P.; Hulscher, K. & Gogolla, M. (2005), 'From UML Models to Graph Transformation Systems', *Electronic Notes in Theoretical Computer Science* **127**(4), 17-33.