Operational Semantics for Object-Process Methodology

Valeria Perelman

Operational Semantics for Object-Process Methodology

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Valeria Perelman

Submitted to the Senate of
the Technion — Israel Institute of Technology
Kislev 5772 Haifa November 2011

The research thesis was done under the supervision of Prof. Dov Dori in the Industrial Engineering and Management Department.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Model-based engineering approaches are increasingly adopted for various systems engineering tasks. Leading characteristics of modeling languages include clarity, expressiveness and comprehension. Exact semantics of the modeling language used in a model-based framework is critical for a successful system development process. As some of the characteristics contradict each other, designing a "good" modeling language is a complex task. Still, an important precondition for acceptance of a modeling language is that its semantics must be precisely and formally defined.

Object Process Methodology (OPM) is a holistic, integrated model-based approach to systems development. The applicability of the OPM modeling language was studied through modeling of many complex systems from disparate domains, including business processing, real-time systems architecture, web applications development and mobile agents design. Experience with OPM has underlined the need to enrich the language with new constructs. An adverse side effect of the increased OPM expressiveness was that it also became more complex and in some cases ambiguous or undefined.

In this work, we define operational semantics for the core of the OPM language using a clocked transition system (CTS) formalism. The operational semantics consists of an execution framework and a set of transition rules. The principles and rules underlying this framework provide for determining the timing of transitions to be taken in a system modeled in OPM. The set of transition rules, adjusted to the OPM rules, describe all the possible changes in the system state based on the current state of the system and the set of its inputs.

Similar works defining formal operational semantics include Statecharts by David Harel, formalizing UML Statecharts with combined graph-grammar and model transition system (MTS) by Varro et al., and formalizing activ-

ity diagrams for workflow models by translating the subject model into a format appropriate for a model checker.

Well-defined operational semantics enables extending OPM with a wide range of testing tools, including model-based simulation, execution and verification, which can employ the theoretical executable framework developed in this work.

As a solid proof of concept, we have developed an OPM-to-SMV (Symbolic Model Verification) translation tool for models in the domain of Molecular Biology (MB), based on the OPM-CTS framework principles and a subset of the transition rules. Using this tool, a holistic OPM model describing both research hypothesis and facts from state-of-the-art MB papers can be translated into an SMV verification tool. The generated SMV model can be verified against specifications, based on information found in MB research papers and manually inserted into the SMV tool. The verification process helps to reveal possible inconsistencies across the MB papers and hypotheses they express as they are all specified in the unifying OPM model.

Abbreviations and Notations

ASL	Action Specification Language
CCS	Calculus of Communicating Systems
CTL	Computation Tree Logic
CTS	Clocked Transition System
LOTOS	Language of Temporal Ordering Specifications
LTL	Linear Temporal Logic
MB	Molecular Biology
MDA	Model-Driven Architecture
MDD	Model-Driven Development
ML	Modeling Language
MOF	Meta-Object Facility
MTL	Metrics Temporal Logic
OCL	Object Constraint Language
OOP	Object Oriented Paradigm
OPCAT	Object-Process CASE Tool
OPD	Object-Process Diagram
OPL	Object-Process Language
OPM	Object-Process Methodology
PTL	Propositional Temporal Logic
PUML	Precise UML
RTL	Real Time Logic
SMV	Symbolic Model Verification
SysML	Systems Modeling Language
T/OPM	a temporal extension of the Object Modeling Technique
TLA	Temporal Logic of Actions
TPA	Timed Process Algebras
TTS	Timed Transition System
UML	Unified Modeling Language
xUML	Executable UML

Chapter 1

Introduction

Developing complex products requires collaboration of multidisciplinary teams starting from the initial systems engineering phases. It is widely accepted that a conceptual modeling process that starts early on during the system life cycle has a great impact on the systems engineering process success. The conceptual model describes the functionality of the system to be developed, defines system boundaries, its relation with the environment, and its high level architecture. It comprises the basis for system-level comprehension by different stakeholders, such as systems engineering managers, project managers, architects, and multidisciplinary teams carrying out the project.

Although the major usage of conceptual modeling is during the early engineering phases, keeping the conceptual model updated is essential, as it increases system reusability, providing the system view during its evolution and allowing test of newly introduced changes to the system at the conceptual level before their detailed design and implementation. The cost of problems detected during or after detailed design is prohibitively high, since this may cause expensive backtracking to earlier phases and further redesign. Thus, there is a need to detect problems starting at early stages of the system development when detailed design does not yet exist.

Figure 1.1 describes a representative set of testing approaches used during the systems engineering process, the diagram is based on [59]. The horizontal axis represents the earliest phases at which the techniques can be invoked, whereas the vertical axis - the level of formality and reliability of the techniques. Ellipses in the diagram represent the different testing approaches, such as questionnaires and more detailed checklists.

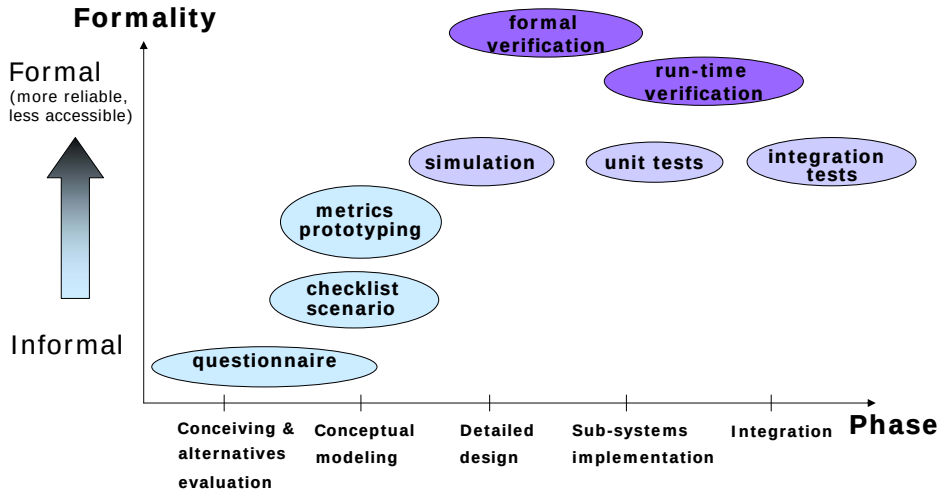


Figure 1.1: Testing Approaches

Paradoxically, despite the need to reveal system problems shortly after their appearance, currently existing reliable and more formal testing approaches, including simulation, model verification and run-time verification, do not operate on the conceptual model, but on the detailed design at a later stage or even at the implementation level. Moreover, most of the existing formal testing approaches and grounded on them tools, such as Modelica and Simulink, require highly experienced professionals to carry them out and to analyze the results, making them too technical to be used during the conceptual modeling by the wide range of system stakeholders.

A conceptual model described using formal unambiguous visual modeling notation will make it possible to construct simulation and testing of its functional and partially non-functional requirements, visualize possible system scenarios, and check some of the system's quantitative and qualitative characteristics. The notation should include refinement, composition and decomposition mechanisms that are appropriate for complex systems. It needs to be implementation-independent in order to focus on the problem domain rather than on the solution domain, and it should be useful as a communication language among multidisciplinary teams.

1.1 Model-Driven Development

Model-Driven Development (MDD) is an approach to software-intensive systems engineering, in which a model becomes the backbone of the whole development process [26]. The approach deals with complexity of systems under development employing graphical modeling methods. It requires keeping up-to-date comprehensive documentation, which enables communicating the solution and the history of changes to multiple stakeholders of the project. The improved communication increases involvement of all the stakeholders in the development process, which contributes to reducing project risks.

At the beginning of the MDD process, the engineering activity centers around the model, and "designing" and "modeling" are synonymous. During later stages of the process, the model serves as a reference and troubleshooting tool. In a genuine MDD process, a model provides value and contributes to the entire project life cycle, but in order to be useful, it must be maintained up-to-date.

Three main architectural aspects that must be defined and documented during the earliest stages of the MDD process are the system boundaries, the solution structure, and the conceptual model defining in detail how the system components will fulfill the required system functionality. Analysis of system models during early MDD phases is a basis for checking the feasibility of the solution and a potential of the future system to satisfy its requirements. It may reveal system bottlenecks and identify trends of various quality attributes, such as system modularity, reliability, and fault-tolerance. A study [30] has shown that essential trends evaluation could be achieved based solely on the system's architecture. The researchers investigated ready-made systems, which were evaluated based solely on their architecture. The trends detected through the analysis were significant and could be useful if they were performed during the initial system architecting phase. A survey of the major architecture evaluation methods can be found in [32].

MDD approaches have drawn much interest by developers of software-intensive systems in industry. Recently, efforts have been made to investigate ways to adopt model-driven systems engineering principles for development of systems that combine both hardware and software components.

Selection of an appropriate conceptual modeling language for the early

development phases within MDD is critical. In MDD of software-intensive systems, a subset of UML diagrams, implementing the Object Oriented (OO) paradigm, is commonly used as a modeling standard. In the case of hybrid systems, the selection may vary. An appropriate conceptual modeling language should:

- provide effective abstraction mechanisms that can hide unimportant details at high-level system views, while keeping all the necessary information within the model. This abstraction addresses the requirement of MDD to deal with complexity of modern systems;

- be highly expressive, enabling description of functional, behavioral and structural aspects of a general system, involving humans and having hardware and software components;

- be comprehensive and easily accessible to enable effective communication among all the stakeholders of MDD;

- be mathematically well-grounded, with formal operational semantics. Formal operational semantics is needed to keep model dynamics clearly defined. Well-defined operational semantics will also enable construction of tool support for model analysis, which is essential in the earliest MDD phases.

1.2 Research Goal and Objectives

OPM as a modeling language has effective abstraction mechanisms, as data is organized hierarchically with a root diagram providing the most abstract view of the system and specifying system boundaries and interaction with external entities and systems. The language has a compact vocabulary of symbols and the same symbols are used in all the model diagrams, which are all of the same type. Despite this, OPM is highly expressive, so structural, functional and behavioral system aspects can be shown in the same diagram. OPM applicability has been studied in many disparate domains, including business processing [15], real-time systems architecture [49], web applications development [55] and mobile agents design [63]. OPCAT [1], which is an OPM CASE Tool, supports, among other useful features, model

animation, described in [71], which can be used to debug control flows of OPM models.

Enhancement of OPM with formal operational semantics will provide a basis for developing an OPM model-based analysis and evaluation framework. Analysis and evaluation at the level of conceptual model will help reduce risks related to the development of error-prone complex systems early in the development process, saving both financial and time resources. It will also make the modeling process more focused on the system requirements and help identify potential system evolution directions. Finally, it will help to predict possible changes to the design, following post-prescriptive analysis.

The goal of the research is to develop and evaluate an executable OPM model-based framework as part of a Model Driven Systems Engineering process to be used during the system conceptualization phase.

In [47], OPM/T modeling language extending OPM to be suitable to specify reactive and real-time systems was presented. The language syntax was defined using a set of production rules of Object-Process-Language (OPL). However, the dynamics of OPM/T was not defined formally, but could be derived from the specified compiler translating the OPL phrases into C++ code. OPM/T has contributed to the development of OPM as a modeling language and methodology.

OPM/T can be seen as the first attempt to provide formal operational semantics for the extended OPM language. The definitions in the OPM/T semantics did not account for object instances and for structural links between instances, similar to Petri-Nets, which have tokens without links between them. Formal rules expressed through implementation of the compiler for translating OPL into C++ do not provide an adequate framework on which OPM operational semantics can be easily changed, neither do they comprise a set of formal specifications that can be used for further development of model-based verification and execution methods.

Inspired by the work on OPM/T and learning lessons from it, in this research, an executable OPM operational semantics framework has been developed, and evaluated using formal specification methods. This includes specifying the OPM operational life cycle and the execution principles. In addition, the OPM operational rules were extended to provide OPM instances management, defining a formal typing system and basic control flow

operations that take into consideration the population of instances and the links among them. New synchronization mechanisms were also developed.

The research objectives were the following:

- Defining advanced OPM language mechanisms to enable manipulation of object/process and link instances;
- Defining formal OPM operational semantics; and
- Evaluating the framework's applicability using a case study from Molecular Systems Biology Domain.

Chapter 2

Theoretical Background

The research focuses on formalization of operational semantics for OPM as a general systems conceptual modeling language. In this chapter, theoretical background related to existing computing systems, is presented. We describe formal models and approaches and their usage in operational semantics for modeling languages (MLs). Then, background related to existing conceptual modeling languages and work in the field of model-based execution and verification is overviewed.

2.1 Systems Classification

Computing systems are commonly classified as *transitional*, *reactive*, *real-time*, and *hybrid* [37]. *Transitional systems*, the most well-studied class, can be defined as set of relations between input and output values. While the processing of the system can be non-deterministic, the only important states in a transitional system are the initial and the final ones. In the initial state an input value is inserted, whereas the appropriate output value is generated in the final state.

Reactive systems can be used for a wide spectrum of real-world systems. They describe a continuous interaction among the system and its environment. A system generating the character "**b**" to all the inputs until it receives the first character "**a**" as its input and starting from this point generating the character "**c**" is an example for a system that can be specified as reactive system and cannot be specified as a transitional one. The

characteristic that differentiates a reactive system from a transitional is that the reactive system captures non-quantitative temporal precedence among events. Following is another example emphasizing this property: "**after a mail was sent, finally the mail will be received**".

Real-time systems were developed to describe systems having soft-time and hard-time requirements. A controller system in a submarine is an example of a real-time system. The controller has a hard-time requirement on the time within which the submarine must dive after an enemy radar was detected. A real-time system is an extension of a reactive system with a metric aspect of time. An example of a specification that can be defined through a real-time system, is: "**after a mail was sent, within 5 minutes, the mail must be received**". In real-time systems, the time advances discretely using a global conceptual clock. In [48], Peleg and Dori have presented OPM/T as an OPM modeling language extension for modeling reactive and real-time systems.

Hybrid systems enable inclusion of continuous components in a real-time system. Such capabilities can be used to describe state variables that depending on physical rules, and, as a consequence, continuously change. An example is a cellular battery capacity, which changes continuously.

2.2 Formal Computational Models

Various formalisms have been developed to address different types of the computational systems. These are used to specify the systems and verify different properties of these systems. Some of formalisms have been used to define operational semantics of high-level conceptual modeling languages, such as Activity diagrams and Statecharts. The following computational models and families of formalisms is only partial; additional ones can be found in [7].

Linear Propositional Temporal Logic (PTL), which is based on the theory of natural numbers with linear order and monadic predicates, has been demonstrated as a working tool for modeling and verification of reactive systems [35], [53]. *Fair transition systems* (FTS), shown in [22], are appropriate for specifying reactive systems as well. Many tools implement FTS verification methods.

Process Algebras is formalism that provides constructs to specify sequen-

tial and concurrent processes, events, actions, and their delays for reactive systems. Calculus of Communicating Systems (CCS) [46] and the Language of Temporal Ordering Specifications (LOTOS) [12] are two example formalisms based on process algebras. Process algebras formalism was used in [36] and [66] to provide operational semantics for Statecharts, a visual modeling language extending FTA (final transition automaton) with hierarchy, concurrency, communication, and priorities. Precisely defining this modeling language semantics is extremely challenging. The semantics also has several dialects, as overviewed in [67].

Timed Process Algebras (TPA) - is an extension of process algebras formalism enabling specification of real-time systems. A brief overview of the formalism can be found in [41], which overviews the rich set of existing approaches and provides a general "cook book", describing the principal questions to be answered while designing a new logic belonging to the family of TPA.

Real Time Logics (RTL) includes a rich family of formalisms, which were designed to express timing properties of real-time systems [28], [23]. In [8], a unified framework for the study of formalisms proposed to specify real-time logics was presented. Based on this framework, the formalisms were classified according to their complexity and effectiveness. While most of the formalisms are appropriate for specifying real-time systems, most of them were shown to be undecidable. In particular, this work has proven that the satisfiability problem for *metrics temporal logic (MTL)* is EXPSPACE-complete.

MTL is a timed temporal logic and an extension of *Linear Temporal Logic (LTL)* [13]. Temporal logic in MTL is adapted to enable reasoning of real-time properties using a time-bounded version of temporal operators, where for example, $\diamond_{\leq c}$ means "*eventually, within time c*". *Timed transition systems (TTS)* are transition systems designed to enable real-time systems specification. In [24], TTS models are presented in two possible styles: time-bounded (MTL-like) and time-referenced, where explicit special clock variables are defined, and a subset of clocks can be reset during any transition. These variables can also be used in transition guards. A proof methodology is provided in this work for verification of both time-bounded style and explicit-clock style models.

A *Clock Transition System (CTS)* [31] defines a natural style of real-

time properties specification using an extended transition system. CTS extends a transition system with real variables to model real-time clocks. In CTS, numerous clocks can be defined and reset within regular transitions, a special tick-transition is used to advance timers, and one global clock, the master clock, which cannot be reset within a regular transition, is used to calculate absolute time in the system. In contrast to the time-bounded semantics, CTS semantics is more natural, as timers are just another kind of state variables in the transition system, and they can be manipulated using the same mechanisms used for regular state variables. A second advantage of such semantics is that it enables reuse of verification tools designed for reactive systems. In [11], CTS was used to model and verify a railroad crossing benchmark problem. Formalizing activity diagrams for workflow models by translating the subject model into CTS in a format appropriate for a model checker was described in [18].

2.3 Conceptual Modeling Languages

Despite the advantages of analyzing systems during their earliest development stages, most of the currently used simulation, verification and validation techniques are done later and at lower system levels. For instance, hardware circuits [27], software coding [25], and formal verification are done at the algorithmic level [57]. Model checking tools and theorem provers [60], [38] are traditionally complex, requiring formal specification of both the model under inspection and of the requirements, thus they are typically used for the system's hardware parts and for the close-to-hardware software modules, such as drivers, for which verification tools with automatic support are highly developed. MoDe [56] is a system-level design methodology that defines qualitative analysis using formal verification methods. However, it works on a limited set of system qualitative characteristics, which includes responsiveness (timing that meets the requirements) and modularity.

The Z language [62], which is based on the lambda calculus theory, axiomatic set theory, and first order logic, was proposed to specify computational systems formally. Object-Z [61], an extension of Z, provides modeling in an object-oriented style for real-time systems. In [69], a compositional verification of Object-Z specifications using LTL was described. Like languages used in model checking, both Z and its extension require a formal

mathematical definition of the system model and of the requirements, and is thus hardly appropriate for the MDD system conceptualization phase.

A runtime verification framework for software systems specified by statecharts was developed in [52], where propositional temporal logic is extended to fit statechart constructs. The executable assertions are derived from the formal models, then runtime verification, based on these assertions, is performed in appropriate phases of the software development.

Unified Modeling Language(UML) is the OMG standard modeling language for specifying and designing software systems. Its structural semantics was formally defined via Meta-Object Facility (MOF) [45] using a metamodeling approach and Object Constraint Language(OCL). To achieve clear dynamic and behavioral semantics of models, UML has been extended by the Action Specification Language(ASL), [5], [68]. A set of actions defined by ASL is based on a limited set of actions specified by the precise UML(PUML), group [4], which worked on the discovery of the UML semantics ambiguities. The efforts of this group were incorporated into UML 2.0 [44], [43]. In [70], a formal action semantics for a UML action language was defined.

Precise action semantics of UML enabled development of UML model execution (xUML) [39], which specifies a deterministic system using a profile of UML, the ASL language, and OCL. xUML models are highly testable, and different tools and research prototypes supporting xUML simulation and verification, such as Bridgepoint [54] and Kabira Action Semantics [29], exist.

Application of verification techniques in the context of a model-driven architecture(MDA) using xUML models is explored in [19]. The verification process is based on the Temporal Logic of Actions (TLA), which was originally defined by Lamport [33]. In [65], assertion-based verification for xUML models was described. This type of verification uses assertions, i.e., definitions of the required system properties, and it checks the model against these assertions. Static assertion-based verification for xUML was defined in [34].

Another work based on the xUML models was implemented in the iUMLite [2] CASE tool. The tool supports model-based simulation of an xUML model of the system, including initial population of objects and the status of the signal queue and the test methods. iUMLite generates simulation code

based on all these inputs and supports running the code with a debugger.

The major criticism of xUML relates to its formalism and coding required during the definition of the model views. UML multiple views also make it difficult to clearly understand the connections among the generated simulation code and the source diagrams.

Modelica [9] is a modeling language and framework for systems engineering. In Modelica, the users can specify a system using ready-made graphical components or specify their own components using a C++ like programming language to define physical and logical dependencies among states of various components. Then, Modelica generates a simulator for the specified system model, which can be invoked and provide data on the change of different system properties during run-time. A main criticism of this modeling framework is that it requires programming skills.

SysML [42] is another modeling language providing solution to model-driven engineering of general systems. The language was developed based on UML, with reduction of the number of possible views from 13 to 9 and with invention of two new modeling diagrams, the parametric diagram (specifying engineering formulas and assertions) and the requirements diagram. In SysML, all the views are separated into four major pillars, defining system requirements, system structure, system dynamics, and system parameters. Efforts are made to keep all the pillars linked, in order to establish relations among requirements, components implementing the requirements, dynamics among the components implementing the requirements, and finally assertions in the parametrics diagrams implementing the requirements and their relations to the attributes of the involved components. No research related to formalization of SysML was found. A comparison between OPM and SysML was performed in [21], [20]. It was stated that no one of the languages is the ultimatively better choice and that the languages can complement each other.

In [58], ModelicaML was presented as a framework for supporting the Model-Based Systems Engineering paradigm. ModelicaML is defined as a graphical notation of Modelica, profiling SysML and extending it with a new Simulation diagram. ModelicaML benefits from the modeling expressiveness of SysML and simulation capabilities of Modelica. The language enables description of time-continuous and event-based systems that include both hardware and software components. It supports all the Mod-

elica constructs, including modeling with equations to specify engineering constraints. Through Simulation diagrams, simulation parameters and results are shown and can be reused.

The semantics of SysML as a profile of UML is far from being precisely specified and its formalization may require too much effort since the language is very complex, holding many different views and symbols. However, Modelica requires clear non-ambiguous definitions. This inconsistency between the state of the operational semantics of SysML and the requirement of precise operational semantics of Modelica has implication on ModelicaML. This criticism of ModelicaML and possible ways to further develop this framework are discussed in [64].

Bibliovich [10] has carried out research aiming to formalize OPM structural semantics. He employed graph-grammar rules and some additional algorithms with coded-in rules to check the validity of the model structure. As noted, this work formally defines the operational semantics for the core of the OPM language, defined next, using a clocked transition system (CTS) [31]. Few advanced OPM features are excluded from this formalism: exception mechanisms defined in [50], the metamodel reflection mechanism proposed as part of advanced features in this work, and OPM resources, as described in Section II-E . From now on, we refer to the remaining parts of the OPM language as the "OPM Core". The applicability of this CTS-based OPM operational semantics formalism was explored using case study from the field of Molecular Biology Systems. In [51], we have described the framework, including the development of a verification tool based on the operational rules as work-in-progress.

Various levels of modeling language semantics are acceptable in the systems engineering field [17]. In this work, the requirement-level semantics was adopted. This level characterizes early stages of a design and ignores technical limitations regarding processing of the system inputs. It assumes that the system is capable of reacting to any incoming events infinitely fast. In contrast, the implementation level emphasizes possible communication disturbances and reactions of the system to events with some delay. We use the requirement-level semantics as it is appropriate for the primary use of OPM during the system conceptualization stage.

Chapter 3

Object Process Methodology

Object Process Methodology, OPM, [16] is a holistic approach to specifying systems. The method integrates structural, functional and behavioral aspects of a system in a single, unified model, expressed bi-modally in equivalent graphics and text with built-in refinement-abstraction mechanism.

An OPM model consists of a tree of hierarchically-organized diagrams with a root OPD (Object-Process Diagram), called the System Diagram (SD), which defines the most abstract view of the system. The OPM entities are *stateful objects* and *processes*. Graphically, objects and processes are respectively represented by labeled boxes and ellipses. The label inside the shape is the name of the entity stands for. *In-zooming* and *unfolding* of an object or a process are two possible OPM mechanisms to control model complexity. Both of them show details of the refined object/process. Unfolding defines structural relations among the main object or process and lower-level, objects and processes. For processes, in-zooming defines also a partial chronological execution order of the enclosed subprocesses. stands for. *In-zooming* and *unfolding* of an object or a process are two possible OPM mechanisms to control model complexity. Both of them show details of the refined object/process. Unfolding defines structural relations among the main object or process and lower-level, objects and processes. For processes, in-zooming defines also a partial chronological execution order of the enclosed subprocesses.

An in-zoomed process consists of an enlarged process ellipse, a set of local objects and processes located inside the process ellipse and other objects and processes located outside the main process ellipse and connected to the

main process. Figure 3.1 shows an OPD in which a **Supply Managing** process is zoomed into. The process has one local object – **Selected Supplier**, and three sub-processes: **Supplier Selecting**, **Ordering**, and **Shipping**. External to the main process are the following objects: **Order**, **Item**, **Supplier**, **Requested Item**, **Approval**, **Supply Manager**, and **Ordering Details**.

- The **Supplier Selecting** creates **Selected Supplier**
- Then, **Ordering** uses the **Selected Supplier** to create the order **Approval**.
- Finally, **Shipping**, handled by the **Supply Manager** and enabled by the external **Requested Item**, creates **Ordering Details**.

The procedural links that connect objects with processes will be described later on. Other links such as those connecting **Order** with **Item** and **Item** with **Supplier** are general *structural links* between two objects. Graphically, a structural link is a solid line, directed in one or two directions. Optional multiplicity indicates how many instances may be connected across an instance of a structural relation. A multiplicity is written as an expression evaluated to a range of natural numbers. A multiplicity at one end of a relation specifies for each instance of an object at the opposite end, that there must be that many instances at the near end. In the example, **Item** has relation to many **Suppliers**, as the character *"m"* stands for the cardinality "many". If cardinality is not explicitly noted, the default is exactly one. Here, each **Order** includes exactly one **Item**. Each OPD other than SD is created by refining a single *main thing* (process or object) by zooming into the thing or unfolding it. By construction, the main thing must appear in the direct parent OPD (the source OPD). The OPDs are organized in a tree structure with an abstracting/refining relation between a parent and its children.

Figure 3.2 shows three diagrams of the same OPM model. Diagram (**a**) is the SD, showing the main function of the system - **Servicing**. The types of links used in SD and connecting **Servicing** process with **Lift**, **User** and **Lift Request** are effect, agent and consumption links, respectively. At this high level of abstraction, the effect link from **Servicing** affects (changes state of) **Lift**, **User** handles **Servicing**, and a **Lift Request** triggers **Servicing**. Diagrams (**b**) and (**c**) show two possible usages of the *unfolding* mechanism, as explained next. Unfolding is used in the following cases:

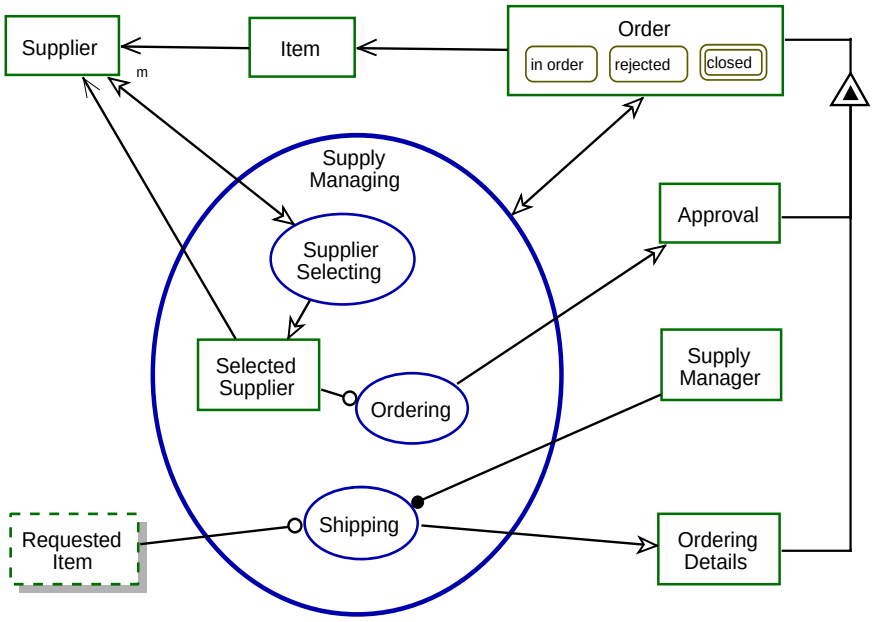


Figure 3.1: Supply Managing In-zoomed

- The new OPD refines structural relations of a high-level object with lower-level objects, such as its parts or attributes. In diagram (**c**), **Lift** is unfolded, showing that it *consists of* **Cabin** and **Engine** and *characterized* by its **Current Position**. A link with a black triangle stands for an aggregation-participation relation among objects or among processes, whereas a link with a black triangle inside a bigger white triangle stands for the *exhibition-characterization* relation.

- The new OPD refines a process, whose subprocesses have no well-defined partial order. In this case, all the subprocesses are event-driven, and their invocation is enabled during the time at which the main process is active. For example, in diagram (**b**), **Lift** can be called by **User** from any floor. Depending on the order of calls, **Lift** will perform **Moving Up** or **Moving Down**. However, the order of execution among the **Servicing** subprocesses cannot be predefined statically in the model. Hence, the diagram defines only structural composition

relation among **Servicing** and the subprocesses that compose it, **Moving Up** and **Moving Down**.

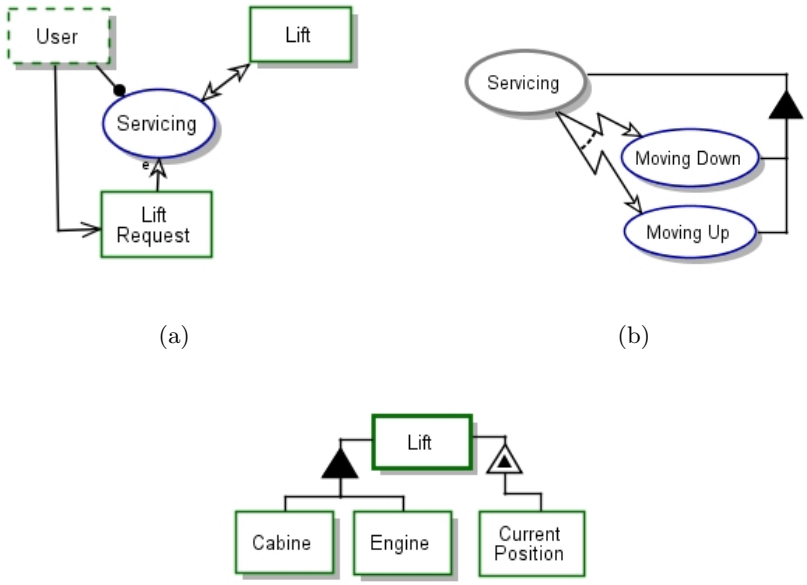


Figure 3.2: Lift system - the tree of Object-Process Diagrams (OPDs). (a) SD - System Diagram - the root diagram. (b) Unfolding of Servicing. (c) Lift unfolded.

3.1 Object States and Instances

At any point in time, an *object* is associated with optional (zero or more) *instances*. Each instance is at a specific single *state* from the finite set of states defined by its object class (often shortly called just "object"), or it has a specific value from the range of possible values defined for the object in case the object is an attribute. The set of object states is the union of disjoint subsets fully covering the space of all the possible object values,

where each state is a range of possible object values. In enumerated types there is identity between states and values, as each range includes exactly one value. For example, in Figure 3.3, an **Employee Date of Birth** (DOB) attribute is defined as an object of **Date** type, while the possible value ranges of the **Age Group** attribute are: between 18 and 35, between 36 and 50, and above 50. An operation **Age Group Assigning** (a "method" of **Employee**) assigns the age group based on the DOB.

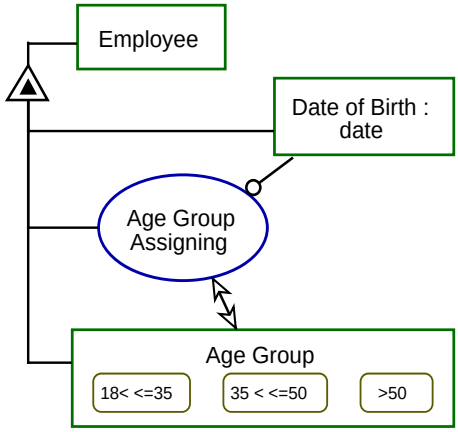


Figure 3.3: An Employee Date of Birth and Age Group Attributes

3.1.1 Process Instances

During an OPM model execution, a process may be invoked simultaneously any number of times. Each process invocation creates a separate *process instance*. A process instance is the only entity in the model that can transform, i.e., create, consume, or change the *state/value* of an instance. A process instance can also use any object instance as an enabler, without transforming it. The type of link connecting a process and an object defines the procedural relation between the process and the object classes, to which the corresponding instances belong. Additionally, a process instance owns its context; this includes its parent process instance, its involved object instances, its local objects, its currently active subprocesses, and its

duration.

3.2 The OPM Typing System

The OPM typing system includes native (built-in) objects, which are grouped into *Numeric*, and *Compound*. *Boolean*, *Char*, *Integer*, *Float* and *Double* belong to the *Numeric* group, whereas *String*, *Time*, *Date*, and aggregates – *Array*, *List*, *Map* and *Set* - to the compound native objects. All the aggregates have additional attribute size of *Integer* type. All the aggregates hold zero-to-many entities of any OPM *Object* subtype; *Map* also associates an object key with each element.

Among the *Numeric* subtypes, there exists the binary relation $\preceq$ defining identity conversion. The same relation exists among objects and processes with an IS_A (a.k.a. generalization/specification) relation. The relation specifies type matching and is reflexive and transitive. It is possible to represent a set of all the model objects and processes as graph nodes with identity conversion relations comprising the graph's directed links. This graph must be acyclic, as a cyclic inheritance is prohibited. Following is the specification of the $\preceq$ relation for the *Numeric* objects:

$$Boolean \preceq Char \preceq Integer \preceq Float \preceq Double$$

According to this specification, *Boolean-to-Integer* conversion is valid, whereas *Integer-to-Boolean* is not.

Any OPM thing has the essence attribute, [16]: "Essence is an attribute of a thing that determines whether the thing is physical or informational. Essence may get one of two values, which are physical or informational." Physical entities have material existence, thus, in the material world, two element are identical if and only if they refer to the same object. All the built-in objects have informational essence.

The OPM typing system includes a set of native processes (operators), which by default take one *tick* (elementary unit) of time. *Identity*, the operator "@", can be invoked on any OPM object, whereas, *IsEqual*, the operator "==", and *Copy*, the operator "=", are defined for built-in objects only. However, these operators can be overloaded for other objects as well.

The *Identity* operator, which returns the object instance, is used to

refer to specific object attributes, as shown later. *IsEqual* and *Copy* accept two OPM objects. *IsEqual* is performed at the level of values identity for primitive types and strings and at the level of reference identity for the built-in aggregation types. *Copy* makes a shallow copy of objects of built-in types. By default, the comparison/copy is shallow, however these operators can be redefined.

The following operators work on *Numeric* objects only: *Adding* (the operator "+"), *Subtracting* (the operator "−"), *Dividing* (the operator "/"), *Remainder* (the operator "%"), *Multiplying* (the operator "∗"), and logical *IsGreater* (the operator ">"), *IsSmaller* (the operator "<"). Operators on *String*: *Concatenation* (the operator "+"), *IsGreater* (the operator ">"), *IsSmaller* (the operator "<"), getting character/object at some place in the *String/Array/Map* (the operator "[]"), accepts an integer parameter specifying the character location, and for *Map* accepts a key of any OPM object type. The operators "+" and "−" are specified for *Time* and *Date* objects to add/remove *Time* from/to *Date*. *Iterate* is specified for aggregate objects. It goes over all the elements in the aggregation (its syntax is defined later). Table 2 in the Appendix A summarizes all the native OPM processes.

In some cases, if a process has more than one object of the same type and the process is asymmetric, (for example, Dividing) it is important to differentiate among the object instances. The difference must be provided by both modeler of such a process and by the user of the process. This problem is resolved by using explicit roles on the links connecting objects with the process. This semantics is described in the Section F (rule *R5*).

3.3 Resources

A *resource* is an object with a measurement units attribute and a utilization function. The utilization function is attached to an effect link connecting the resource to the utilizing process. For example, in Figure 3.4, **CPU utilization** can be measured by percents, and the percent of utilization is denoted along the instrument link between **CPU** and **Data Processing**. The *utilization bound* of a resource can be hard or soft. Hard utilization bound means that the utilization cannot exceed a predefined maximal value, whereas in a soft bound utilization this may be possible. For example, it is impossible

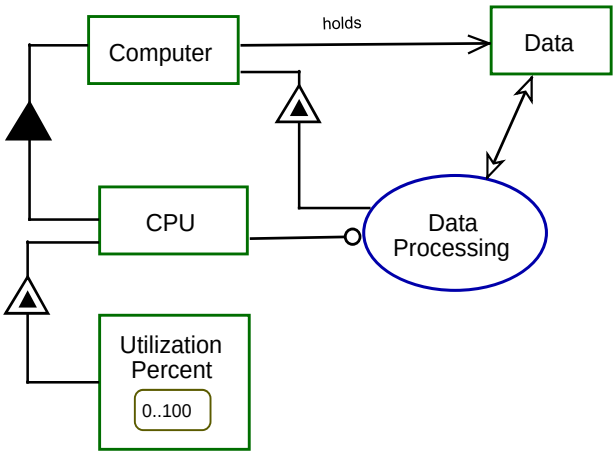


Figure 3.4: Data Processing Diagram

to consume more **Fuel** than the volume of the **Fuel Tank**, but it is possible for a **Person** to be employed more than 100% **Employment**, and this can be measured in units of **FTE** (**Full Time Employment**).

The type of a resource usage defines if the resource returns to the previous state after the usage. For example, the **Computing** process utilizes 10% of the **CPU** power, after the process is finished, the **CPU** power of the **Computer** returns to its previous value, which is 100%, unless another process is still active.

Another characteristic of resource utilization is the *resource recoverability*, which can be total, partial or none. For instance, the resource recoverability of a (non-rechargeable) **Battery** of a **Digital Camera** is partial, as every use of the **Digital Camera** reduces the capacity of **Battery**. The resource recoverability of a **Fuel Tank** is full, as it can be refilled after being depleted.

3.4 Object Transformation

Procedural links are used to define the dynamic relations among processes and objects. They include instrument, condition, agent, effect, consumption, result, and input-output pair links. A result/consumption link stands for creating/destroying an object by the connected process, and it is symbolized

by a directed arrow from/to the process. In Figure 3.1, the **Supplier Selecting** process creates the **Selected Supplier** object; the process is connected to the object with a result link. Instrument and condition links indicate the process enablers, whose values are preserved. In the example above, **CPU** enables **Data Processing** via an instrument link. Graphically, the condition link is like the instrument with a "c" character written inside the blank circle.

The agent links a line ending with a filled-in circle ("black lollipop") which denotes "human-in-the-loop" – the activation of a process and/or involvement of a person in the process. In Figure 3.1, **Supply Manager** is a person who takes part in the process, thus the agent link connects the object representing the agent (person) with the process.

The effect and the input-output pair links indicate that the process changes the value/state of the connected object. Graphically, the link is drawn as a bidirectional arrow connecting an object with a process, or a pair of directed arrows connecting different states of an object with a process. The input-output pair is the refined form of the effect link, showing the specific change of an object by the process. For example, in Figure 3.3, a **Moving Up** process changes the value of the **Lift Location**. In the higher and more abstract view, one can model that **Lift** is affected by **Servicing** using an effect link between **Lift** and **Servicing**, as shown in Figure 3.2 (a). **Moving Up** is one of the subprocesses of **Servicing**, as shown in Figure 3.2 (b).

In Figure 3.5, the effect link is refined into a set of input-output pairs among **Lift Location** states and the **Moving Up** process. The pairs are distinguished using path labels recorded along the links. These labels define paths using entry and exit link from/to states in the same object. Thus, following path **a**, **Moving Up** changes **Lift Location** from **floor 1** to **floor 2**, while following path **b**, the process changes **Location** from **floor 2** to **floor 3**. According to [16], when procedural links that originate from an entity are labeled, the one that must be followed is the one whose label is identical with the label of the procedural link that arrives at the entity.

All the procedural links from an object or a state to a process except for the condition link and the exception/event links have the "wait until" semantics. "Wait until" means that the triggered process checks the preprocess object set - the set of all the objects connected via one of these links to the process being checked. For the process to happen, each object in the preprocess object set must have at least one instance. Moreover, if

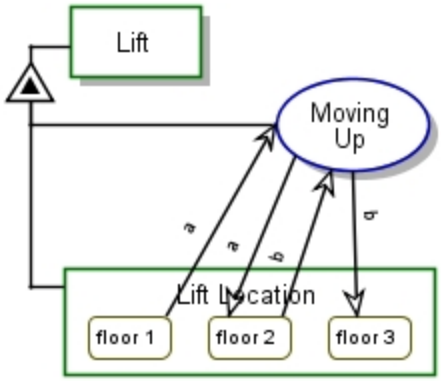


Figure 3.5: Path-labeled input-output link pairs. Path a is combined of floor 1 - Moving Up - floor 2. Path b is combined of floor 2 - Moving Up - floor 3

the link is from a certain state, the instance must be at that state. If this condition does not hold, and the process is non-compound (has no refinement), the process waits until all the objects surrounding the process - the preprocess object set - meet this condition, and only then the process "fires" and becomes active. In contrast, the condition link has a "skip" semantics, which means that the process checks the link only once; if the precondition fails, the process is skipped. Links with the "skip" semantics have higher precedence than "wait until" links.

Logical connectors including n-ary XOR, AND, OR and unary NOT can be invoked on the entry/exit process links/link. Graphically, links to or from disjoint points along the process ellipse mean there is an AND connection among the links. Links connected with doubly/singly dotted arc denote links with an OR/XOR connection among them. A link with a small vertical line crossing the link edge near the process end denotes NOT and means that the object connected to this link must not have an instance in order for the process to happen. The precedence among the logical operators is as follows: NOT, then OR and XOR with the same precedence, then, the lowest is AND.

A process will be invoked only if the enabled incoming links satisfy the logical expression constructed using the logical operators described above.

Some outgoing links are selected from the beginning of the process activation. These outgoing links are connected to the process with AND or have a path label identical to the path label of incoming links connecting the process instance with the same object instance. By the process termination, transformation of objects connected to the outgoing links set must be completed. The link selection is non-deterministic and must satisfy the logical exit expression.

An object can be transformed by a process not only in a discrete manner (at the beginning, at the end or somewhere in the middle of the process activation), but also continuously. For example, a **Gas Filling** process changes the volume of **Gasoline** in the **Gas Tank** over time. Thus, any type of effect link among process and the transformed object means that by the end of the process execution, the instance of the object connected to the transforming process is changed. This change takes a positive amount of time and it can be continuous or discrete. In this work, the assumption is that for non-compound processes that have no time-dependent transformation function, the transformation is complete no later than at the process termination. Consumption happens at the beginning of the process activation and creation by the process termination.

Time-dependent transformation functions, which are typical in systems in general, cannot be described based solely on the rules defined above. For example, in the case of the **Gas Filling** process, there is a period, which is equal to the duration of the process during which the **Tank** lefts its **empty** state and is in transition between the **empty** and the **full** states. There is a volume attribute with a continuous value v that is determined by the equation $v = r * t$, where r is rate in liter/sec units and t is time in sec. There is a correspondence between the value of **Volume** $v = 0$ and the state **empty** of **Tank** and between the value $v = vmax$ of **Volume** and the state **full** of **Tank**. The process **Gas Filling** holds the equation $v = r * t$, continuously changing the value of the **Tank's Volume**. The correspondence between the value of the **Volume** attribute and the states of the **Tank** can be designed as a model constraint. Constraint processes typically hold time-dependent functions, so they must be evaluated after each *tick* of time (explained later) during their lifetime.

OPM refinement mechanisms enable refining procedural links among the refined processes and the refined objects, the following rules illustrated in

Figure 3.6.

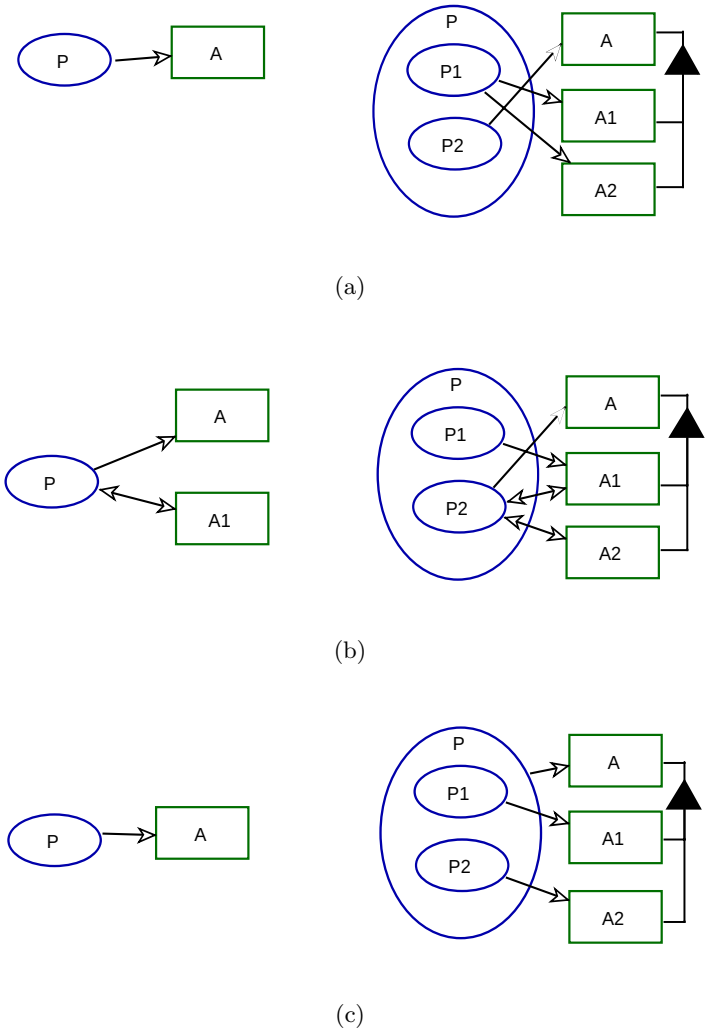


Figure 3.6: A result link refinement. The diagrams appearing in the left side show property suppressions.

The non-formal operational semantics of such refinement is detailed in the following list:

- A process **P** creates, yields (results in) an object **A**. The following cases are possible:

  a) In the refined diagram all the attributes and components of **A** are created or taken from the process **P** input set and assembled into the object **A**. This case is illustrated in Figure 3.6(a). The subprocess **P1** creates all the components of **A** then another subprocess, **P2**, assembles **A**.

  b) One or more instances of objects that are part of **A** exist before the process **P** happens. **P** affects them by making them part of **A**. Then, the instance process **P** will use instances of these objects to create an instance of **A**. In Figure 3.6(b), **P** creates **A**, which is composed of the parts **A1** and **A2**. The process has **A1** in its preprocess object set, and it becomes part of **A**.

  c) There exist subprocesses of **P** that create part(s) of object **A** or set values of one or more of its attributes, but no subprocess creates the object **A** itself. Then, by the end of **P**, an object **A** with relations to all its attributes and parts will be created. In Figure 3.6(c), subprocesses of **P**, **P1** and **P2**, create parts of object **A** then **P** assembles **A**.

- **P** consumes **A**. Any feature (attribute or operation) of a thing T is dependent on T, so cannot live without T. Parts, however, can overlive the whole thing they are composed of. For example, **Name** of a **Person** characterizes the **Person** and after the **Person** object was consumed, it will also be consumed. In contrast, consumption of a biological complex that is composed of a set of molecules won't consume the molecules them selfs, but only the connection among them. Thus, while consumption of an object means automatic consumption of all its attributes, consumption of an object parts must be defined explicitly. In Figure 3.7, the object **A2** is affected by **P** because it looses its relation to the object **A**. **A1** is a part of **A**, which is explicitly consumed by a subprocess of **P**. The object **A3** is consumed as a feature of the object **A**.

  The following cases are possible:

  a) There exist subprocesses that belong to the process **P**, that consume a subset of the object parts, finally a subprocess belonging to **P**

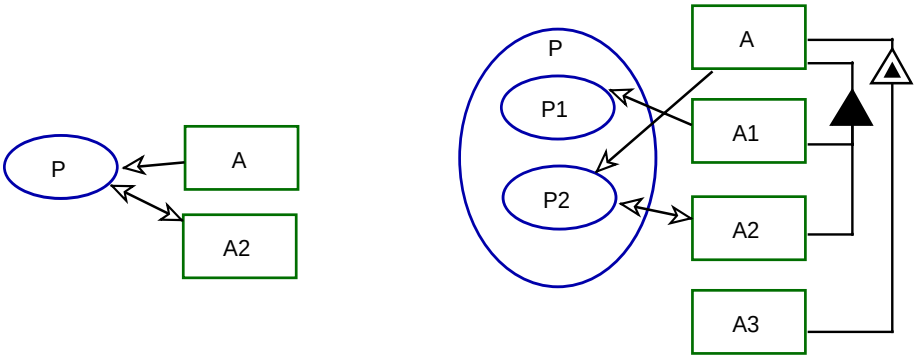


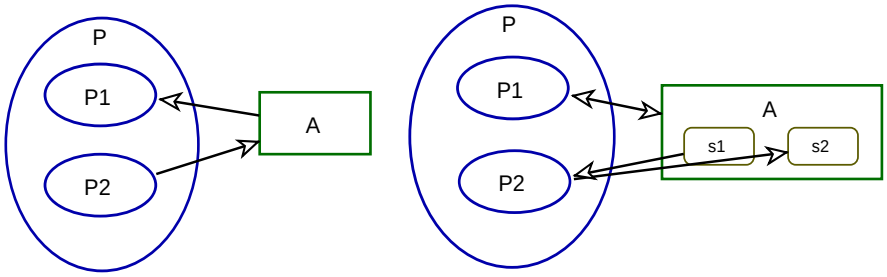
Figure 3.7: A consumption link refinement. The diagram appearing in the left side shows property suppressions.

consumes the object **A**. Then, **A** and all its attributes but only those parts that were explicitly consumed by subprocesses of the process **P**. In Figure 3.7, despite the fact that the object **A** is consumed, its part, the object **A2** is not consumed.

b) There exist subprocesses that belong to **P**, that consume the object components, but there is no subprocess belonging to **P** that consumes the object **A** itself. Then, before the process is terminated, it consumes the object **A**. Suppose that in Figure 3.7, the in-zooming diagram of **P** holds a consumption link from **A** to the process **P** and to its subprocess **P2**. In this case the object **A** will be consumed by the process **P**.

- **P** updates **A** (or sets the object from some state to a specific state **st**). The effect link of a process must be refined if the process is non-compound. The following cases are possible:

  a) There exists a subprocess of **P** that consumes **A**, followed by some other subprocess that creates another object instance of **A** (and the subprocess creates the object in the state **st** or there exists a following subprocess that finally changes the state of the object to the state **st**). The instance is actually replaced by some other instance of the same object class. In Figure 3.8(a), a subprocess of **P**, **P1**, consumes an instance of **A**, then a new instance of the object is created by another



(a) (b)

Figure 3.8: An effect link refinement.

subprocess of **P**, **P2**.

b) Some subprocesses of **P** update the state of **A**. Then, if **P** defined the specific state **st**, the latest subprocess of **P** that transforms **A** must transform it to the **st** state. In Figure 3.8(b), both subprocesses of **P** modify an object instance of **A**.

c) Subprocesses of **P** create/consume or change states of **A**'s parts or attributes.

3.5 OPM Process Invocation Rules

In this section we define rules related to process invocation. The rules are illustrated in Figure 3.9 and Figure 3.10. Figure 3.9 represents an average calculation process, **Average Mfg Year Calculating**, of a fleet of company cars. Figure 3.10, shows a **Signaling System** object unfolded. **Signaling System** consists of zero to many **Sensors**, whose states are re-checked continuously through the "constraint" process - **Sensor Testing** attached to the **Signaling System**. The **Test Result** object entering state **Faulty** triggers a **Siren Alarming** process, whose output is the object **Alarm**.

R1 The timeline rule: subprocesses within an in-zoomed process are executed from top to bottom such that the top-most point of each sub-

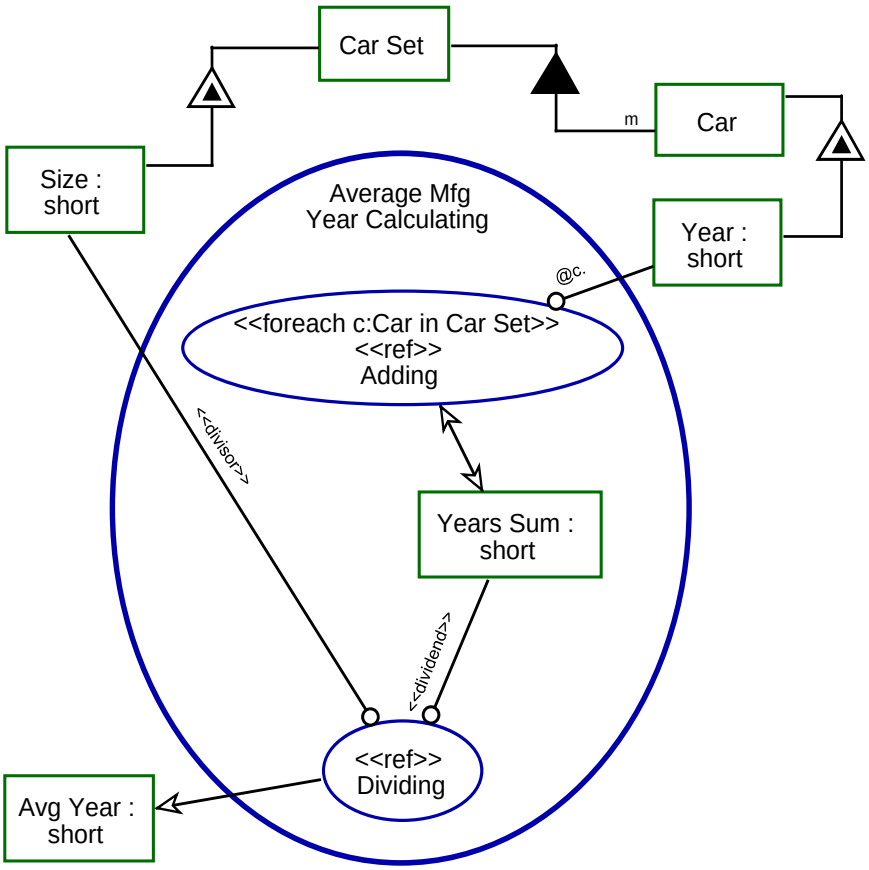


Figure 3.9: Calculating the average year of cars. Process In-zoomed

process ellipse serves as a reference point, so a process whose reference point is higher than its peer starts earlier. If the reference points of two or more processes are at the same height, within a predefined tolerance (of several pixels, depending on the resolution of the screen), these processes start simultaneously. Following the timeline rule, **Average Mfg Year Calculating** process in Figure 3.9 starts with **Adding** of **years** of all the **Car**s, followed by the **Dividing** process.

R2 Flow of control rule: any link from one subprocess of P to another (such as invocation or event) has precedence over the default timeline rule (R1).

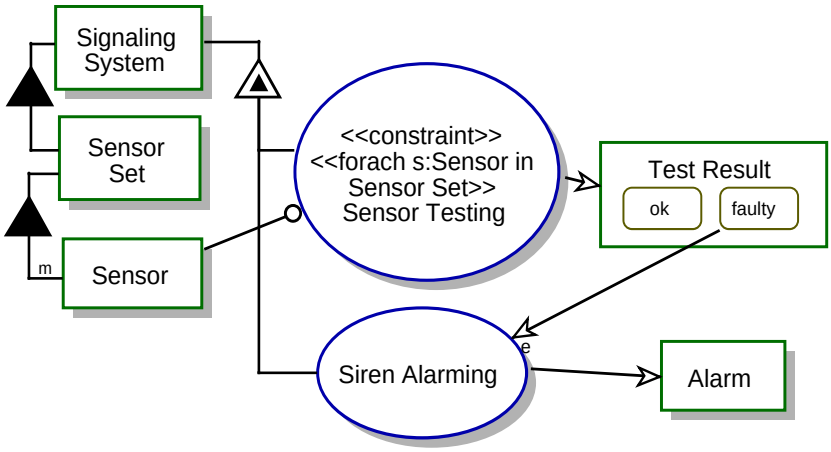


Figure 3.10: Signaling System unfolded

R3 The local objects rule: all the objects that are enclosed by an in-zoomed process P are its local attributes. If no result link connects such an object with any of P's subprocesses, they are created infinitely fast during P's invocation. Their scope is local to the parent process and their lifetime is limited to their lifetime of P. In Figure 3.9, **Years Sum** is an attribute local to the **Average Mfg Year Calculating** process. It is created when the whole process is activated, the **Adding** process modifies it, and it is removed when the whole process finishes, because no explicit result or consumption from any subprocess is connected to it.

R4 The process context rule: the context of the subprocesses of P is partly derived from the context of P, within which they were invoked. Local objects of P can be their instruments, modifiers or results. In Figure 3.9, **Dividing** uses **Years Sum**, an object which is local to the **Average Mfg Year Calculating**.

R5 The constraint rule: A *constraint* is a special type of operation, a process feature, with the keyword "constraint". A characterization relation attaches the constraint to the constrained *thing*. The constraint lifetime is bounded by the lifetime of the thing instance which it character-

izes. A constraint is re-activated in an infinite loop during the lifetime of the thing. A constraint may change one or few of its parameters or may just distinguish a system malfunction during the simulation, depending on the usage. Using constraint mechanism active objects (used in agent-based systems) can be modeled. In the OPD in Figure 3.10, a **Signalizing System** object demonstrates the usage of the constraint mechanism. **Signalizing System** includes a **Sensor Set** that is re-checked continuously through the "constraint" process - **Sensor Testing** attached to the **Signalization System**.

R6 The reference rule: A subprocess can be *referenced* (exposed and reused) using the "ref:path" role, in any context rather than just in a specific diagram. In the expression *ref:path*, ref is the keyword denoting referencing a process, whereas path is the unambiguous path to the process definition. Issues related to this rules follow:

- Type matching of the objects linked to the referenced process and those used in the process definition must be preserved. Type matching is defined in the OPM Type System section.
- If more than one link of the same type connect objects with a referenced process, it is required to differentiate among the objects. This is done using roles on the links both along to the referenced objects and in the diagram, where the process is defined.
- A referenced process is in the context of the parent process enclosing the reference. In Figure 3.9, both **Adding** and **Dividing** are defined in the OPM basic operations library and are referenced in the diagram using the "ref" keyword. The processes are invoked in the model by the **Average Mfg Year Calculating** process and use its context. **Dividing** works on **Years Sum**, which is local to **Average Mfg Year Calculating**. While referencing the **Dividing**, it is important to differ between *dividend* and the *divisor* input objects, as both are connected to the **Dividing** process with instrument links.

R7 The iterative invocation rule: a process can be invoked in a loop, using the keyword "foreach", which is written as a role and defines a loop over a set of elements. The format of the expression is described in the next section. In Figure 3.9, the **Adding** is activated in a loop and

executed for each **Car** in the **Car Set**; **@c** holds the value of the iterator c in the current cycle (the Identity operator). The symbol "@" is an operator used to refer the value of an iterator. The dot operator refer to any of its attributes, if the referred object has some other relation with the iterated object, the name of the relation must be used. The specific reference is denoted along the link connecting the object with the process inside the loop.

R8 The invocation by a single event rule: a process linked to more than one event link can be invoked by any one of the events specified by these links. The links can be of the following types: invocation, instrument-event, effect-event and timeout. In Figure 3.10, the **Sensors Testing** process generates the **Result** object. Its state **faulty** is connected to the **Siren Alarming** process with an instrument-event link. The event that triggers this process, is "entering state **Faulty** by any **Result** instance".

R9 The non-interference rule: a subprocess inside an in-zoomed process cannot be the destination for an event link from an object or state outside P, because this amounts to interfere with the order of execution of P.

3.6 OPM Process Commands

Textual elements add semantics to the model. The elements must be well formed to enable an executable modeling language. The text must have highly formal action semantics. In the following example the process **Processing** has a rather complex precondition. Using fully qualified names of the source edges in the process links, we express the following precondition: **C1** at state **1** and either **C2** or **C3** at state **0**, or **C1** at state **0** and **C4** at state **0**. Using formal textual commands enable expressing complex conditions in a compact yet comprehensive way. Note that we use unique paths to the object states to refer to different input links connecting the objects with **Processing**.

We have shown examples of process commands as part of OPM. The defined commands can be attached to any OPM process. Below, we define formally OPM/AL (Action Language) using Backus-Naur Form (BNF). The bold tokens symbolize terminals.

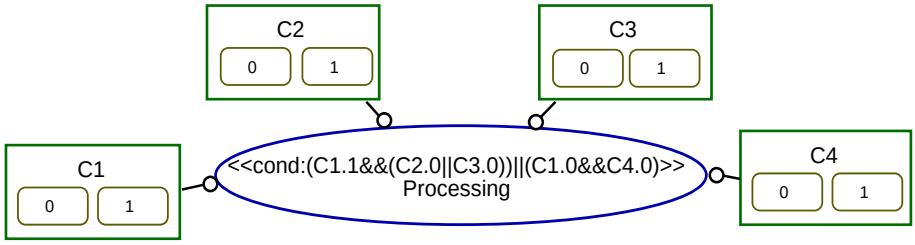


Figure 3.11: OPD showing the Processing process with a condition

$$
\begin{aligned}
&OPMActionsBlock ::= << OPMBlock >> \\
&OPMBlock ::= PathReferenceExpr \mid PathReferenceExpr\textbf{;}OPMBlck1 \mid \\
&OPMBlck1 \mid \epsilon \\
&OPMBlck1 ::= OPMLoopStmt \mid OPMLoopStmt\textbf{;}OPMBlck2 \mid OPMBlck2 \\
&OPMBlck2 ::= OPMGateExpr \mid OPMGateExpr\textbf{;}OPMBlck3 \mid OPMBlck3 \\
&OPMBlck3 ::= OPMMetaDataExpr \mid OPMMetaDataExpr\textbf{;} \\
&OPMConstrBlock \mid OPMConstrBlock \\
&OPMConstrBlock ::= \textbf{constraint} \mid \textbf{constraint ; atomic} \mid \textbf{atomic} \\
&PathExpr ::= \textbf{ref:}SystemPathIdentifier\textbf{.}ObjectPathIdentifier \\
&\mid \textbf{ref:}ObjectPathIdentifier \mid \textbf{ref} \\
&ObjectPathIdentifier ::= ObjectIdentifier\textbf{.}RelIdentifier\textbf{(} \\
&ObjectPathIdentifier\textbf{)} \\
&\mid ObjectIdentifier\textbf{.}ObjectPathIdentifier \mid ObjectIdentifier \\
&OPMMetaDataExpr ::= SystemPathIdentifier\textbf{.}ObjectPathIdentifier \mid \\
&ObjectPathIdentifier \\
&OPMLoopStmt ::= \textbf{foreach } Identifier \textbf{ in } ObjectPathIdentifier \mid \\
&\textbf{foreach } Identifier \textbf{ in } OPMLoopStmt \\
&OPMGateExpr ::= \textbf{wait until:}LogicalExpr \mid \textbf{cond:}LogicalExpr \\
&\mid \textbf{cond:}LogicalExpr \\
&LogicalExpr ::= OpLogicalExpr \mid \textbf{(}LogicalExpr\textbf{)} \mid LogicalExpr \\
&BopLogicalExpr \mid PathIdentifier \\
&Op ::= \textbf{!} \\
&Bop ::= \&\& \mid \| \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (3.1)
\end{aligned}
$$

The root element is the *OPMActionsBlock*, which includes a process reference *PathReferenceExpr*, followed by the iteration statement *OPMLoopStmt*. *OPMGateExpr* defines the process activation condition of "wait until" or "skip" semantics, depending on the first keyword in the expression. The last two elements are *OPMMetaDataExpr* element, which describes thing role and constraint and atomic keywords. The commands in the block are ordered and each command is optional.

3.7 OPM Design by Contract

Design by contract is a software approach founded by Meyer, [40], to create safer program modules with well-defined API that reduces side effects of the program behavior. According to this approach, a method or a class defined in a design by contract language explicitly defines all the required pre-conditions, post-conditions and invariants. Pre-conditions are checked at the method entry point, post-conditions are checked at the method return, and invariants, are checked after/before each one of the method statements.

In OPM, the precondition of a non-compound process can be defined with a combination of condition and instrument links. The postcondition is defined using a combination of a result and effect links and checking the created object or modified object state. An instrument link to a non-compound process is an OPM equivalent to assertion.

3.8 OPM Synchronization Lock Mechanisms

As hardware becomes cheaper and networks grow to be more reliable, modern systems widely use distributed architectures. This leads to the need of modeling distributed systems and providing synchronization abstractions within modeling frameworks. This section is devoted to synchronization issues in OPM.

Since OPM is appropriate for general systems development, the OPM language has non-interleaving concurrent computational model, i.e. parallel processes are executed simultaneously, possibly using physically separate processing devices. Non-interleaving semantics makes modeling of physically distributed systems more realistic and natural.

In non-interleaving computational models, it is required to have a specially-designed synchronization mechanism, such as "Test & Set" [14] to provide mutual exclusion of processes. In OPM, we introduce an atomic mechanism for addressing this issue. The "**atomic**" process tag denotes the process atomicity.

R10 Atomic Process Execution Rule: an atomic process **Lock Acquiring** sets an object of type **Lock** or of an object that is derived from **Lock** from state **0** to **1**, and an atomic process **Lock Releasing** transforms the **Lock** from **1** to **0**. The object **Lock** and the processes **Lock Acquiring** and **Lock Releasing** are features of a process whose subprocesses must be synchronized. The two subprocesses **Lock Acquiring** and **Lock Releasing** are executed via an uniterruptable sequence of commands, not in parallel to or in an interleaving mode with any other subprocesses of the parent process instance.

The Milk Supplying Problem example a couple consisting of a husband and a wife are required to buy exactly one bucket of milk. They have a refrigerator; they can check its content and can leave notes to each other on the refrigerator. Algorithm 1 has no mutual exclusion mechanisms, thus it is

Algorithm 1 Basic symmetric algorithm of Milk Buying for each Person in the Couple

```
if no Milk in the Refrigerator then
  Buy Milk
  Put the MilkBucket into the Refrigerator
end if
```

possible to end up with two milk buckets in the refrigerator. The following sequence represents a legal algorithm execution that finishes with two milk buckets: (1) The husband checks the refrigerator, (2) The wife checks the refrigerator (line 1), (3) The husband buys milk (line 2), (4) The wife buys milk (line 2), (5) The husband puts a milk bucket into the refrigerator (line 3), (6) The wife puts another milk bucket into the refrigerator (line 3).

Figure 3.12 is the OPM implementation of the algorithm. Figure 3.13 demonstrates a simulation result of the algorithm. The picture is taken at the end of the **Milk Buying** process activation. The initial setup included an empty refrigerator. The number in the corner of the **Milk Bucket** object shows

the current number of instances. The initial setting before the **Milk Supply** process activation was zero. The example demonstrates the synchronization problem of the algorithm, where no synchronization mechanism was employed, using OPM model execution.

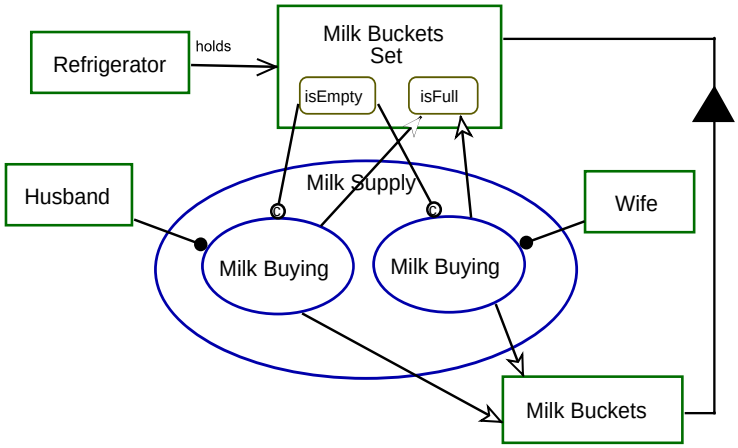


Figure 3.12: OPD showing Milk Supplying Process

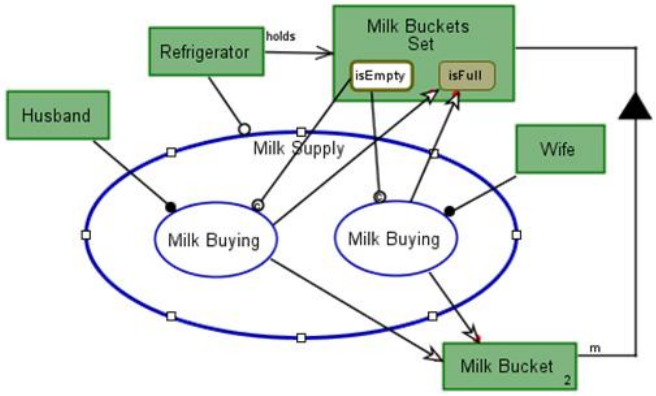


Figure 3.13: OPD showing Milk Supplying Model Animation

The classical anti-symmetric solution to the problem uses two named notes as follows.

Algorithm 2 Milk Buying by Wife

```
Leave Note from Wife
while no Note from Husband do

end while
if no Milk then
  Buy Milk
  Remove Note from Wife
end if
```

Algorithm 3 Milk Buying by Husband

```
Leave Note from Husband
if no Note from Wife then
  if no Milk then
    Buy Milk
    Remove Note from Husband
  end if
end if
```

The algorithm does not satisfy the fairness[1] property, since Wife is more likely to buy the Milk. Moreover, the algorithm preserves a correct behavior only if the executing machine running the algorithm has a single processing device.

Three diagrams that specify the algorithm in OPM are presented in Figure 3.14-Figure 3.16. Figure 3.14 is the root diagram. The two diagrams that follow represent the Wife and Husband algorithms, respectively.

The next solution of the milk buying problem uses "atomic" mechanism that synchronizes access to the object **Lock**. The solution provides mutual exclusion, still no fairness is guaranteed. However, it is possible using the atomic mechanism to construct advanced synchronization mechanisms for supporting the fairness property.

Figure 3.17 shows the OPM solution using the new "atomic" mechanism. The solution is general, it is symmetric for all the participants and correct for more than two participants.

The difference between the "atomic" and the "invariant" mechanisms

[1]Strong fairness is expressed as follows: $\forall f \forall P.$ Pproceeds infinitely often enabled in f, where P denotes a process and f is a computation.

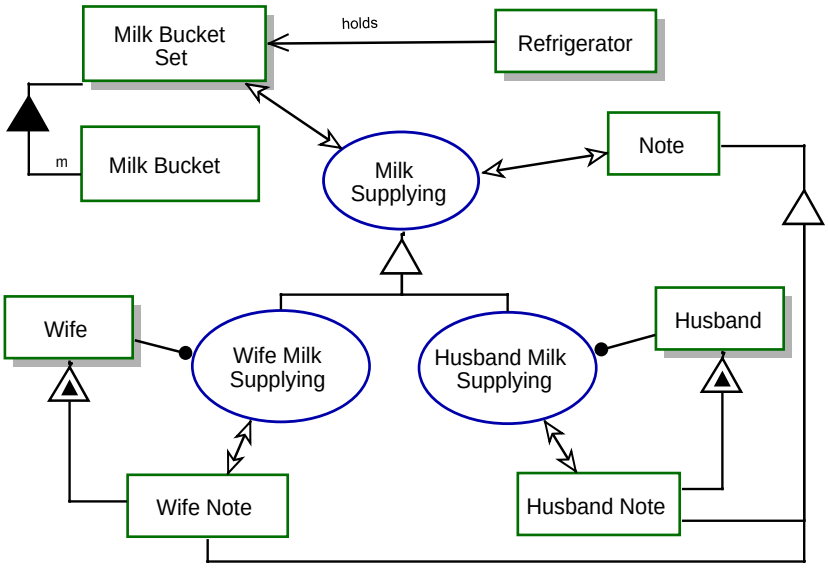


Figure 3.14: An OPD showing the Asymmetric Milk Supplying Process Unfolded

Algorithm 4 Milk Buying using the Lock Mechanism

```
Acquire(Lock)
if no Milk then
  Buy Milk
  Release(Lock)
end if
```

is as follows. The atomic mechanism is used to provide mutual exclusion. In contrast, the invariant mechanism defines that the process requires the object for its execution, but it does not prevent any other process instance from changing or consuming the required object.

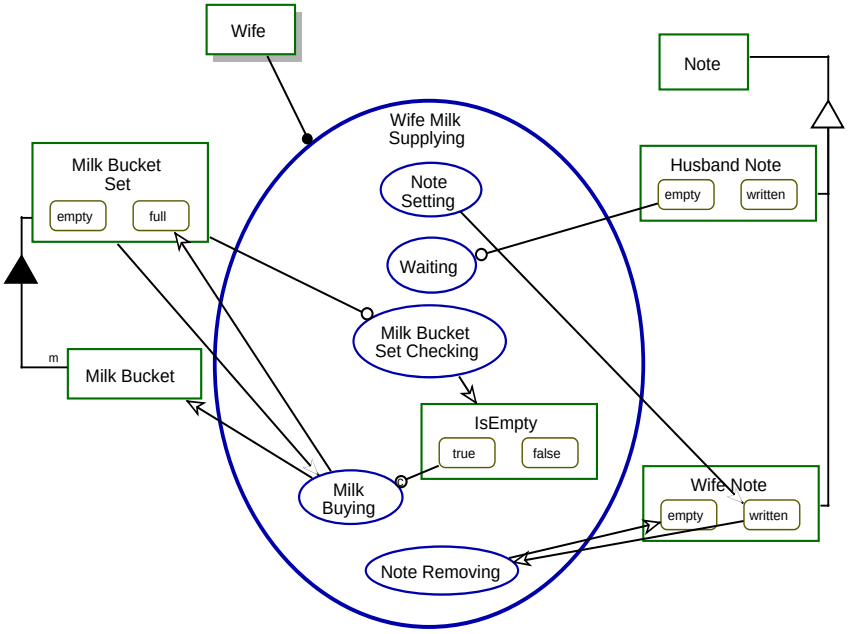


Figure 3.15: OPD showing Asymmetric Milk Supplying - Wife buys the Milk

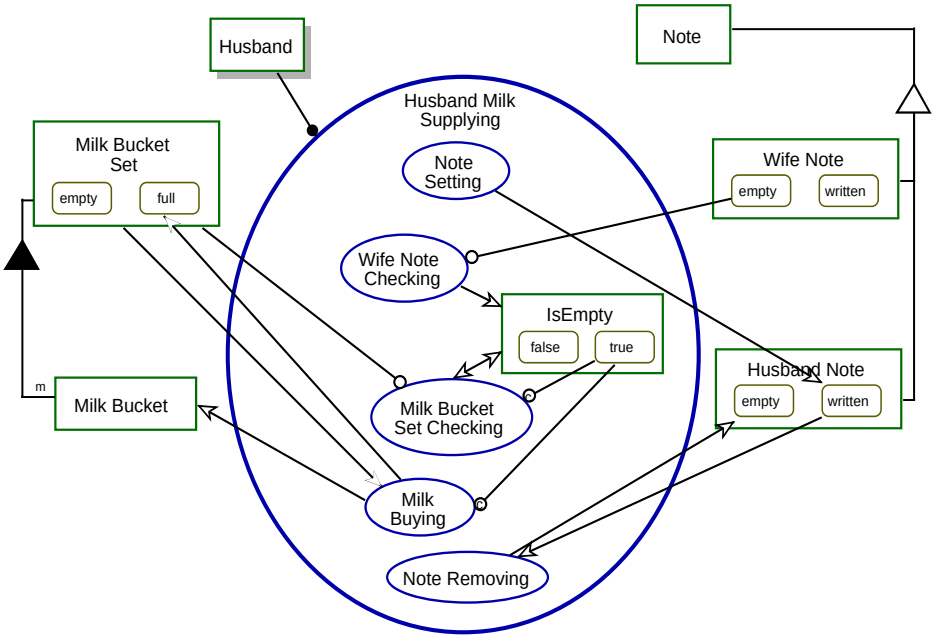


Figure 3.16: OPD showing Asymmetric Milk Supplying - Husband buys the Milk

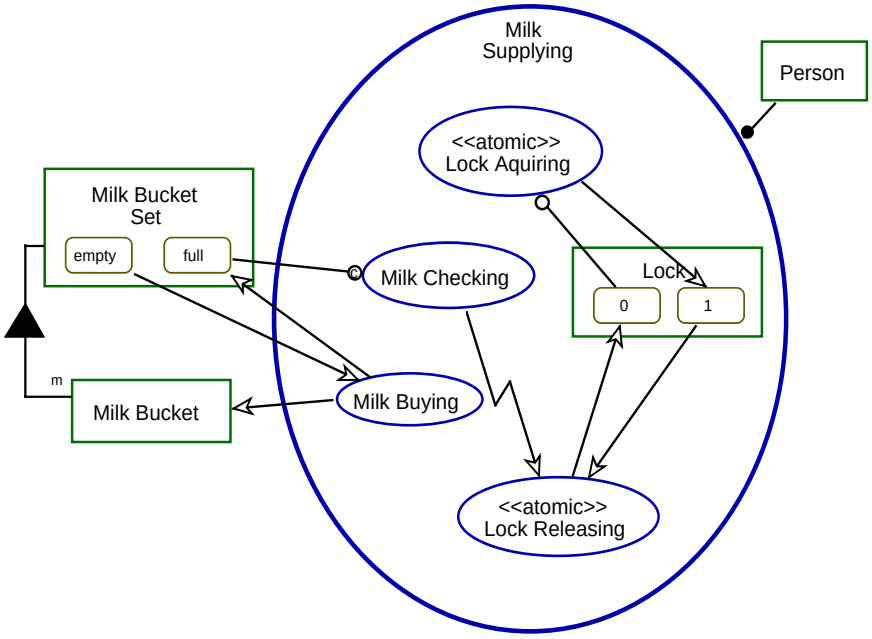


Figure 3.17: Fair Milk Supplying using Mutex

Chapter 4

Defining OPM Core Operational Semantics via Clock Transition Systems

Clock Transition System (CTS) is a computational model for real-time systems that was developed in [31]. Inspired by the timed automata model [6], CTS is more suitable for modeling a rich language such as OPM. In the following definition of CTS, we follow closely the definition in [31].

CTS is an extension of a *transition system* to a clocked transition system. A transition system defines possible moves between states. Formally, it is a tuple $\langle Vars, \Theta, \Upsilon \rangle$. $Vars$ denotes a finite set of state variables, which means that a state of the system is defined by a full valuation of the variables in $Vars$. Θ is a predicate over $Vars$ that defines the initial condition, and Υ is a finite set of transitions. Each transition $\tau \in \Upsilon$, is a function

$$\tau : 2^{Vars} \mapsto 2^{2^{Vars}},$$

i.e. τ maps each state σ into a (possibly empty) set of states $\tau(\sigma) \subseteq 2^{2^{Vars}}$.

A state is denoted by σ, and $\sigma(x)$ for $x \in Vars$ denotes the value of x in state σ. For notational convenience we will write simply x to denote $\sigma(x)$, when it is clear from the context that we refer to a state σ. We denote by x' for $x \in Vars$ the value of x after a transition.

The difference between a transition system and CTS is that the latter includes also variables that represent system clocks. The transition relation

Υ now includes a special transition τ_{tick}, which represents the passage of time. In addition, a CTS includes an element Π, which specifies a condition over the progress of time.

In the next sections, we develop a formal definition of the OPM operational semantics using the CTS formalism. We begin by defining the OPM graph.

4.1 OPM Graph Representation and Construction

Let $Objs$, $Procs$ and $States$ denote sets of object, process, and state nodes, respectively. All the OPM objects and processes are translated into graph nodes in $Objs$ and in $Procs$ sets respectively. All the object states within each object are translated into nodes belonging to the $States$ set.

OPM links can be procedural or structural. Let $StructLinks$ and $Proc\ Links$ denote the sets of structural and procedural links, respectively. Procedural links denote dynamic relations, typically between an object and a process, while structural links define continuous relations between any two objects or any two processes. Exhibition-Characterization is the only relation in which a structural link can connect an object and a process.

A procedural link describes a role of an object in the process to which it is linked. Examples include the role of an instrument that enables a process, consumption and creation of an object instance by a process, change of an object state by the process, event links and condition links. All these types of links are translated into the corresponding edges in the $ProcLinks$ set. Procedural link instances exist for a limited period of time, bounded by the smaller of the lifetimes of the two stateful objects and/or processes at the edges of the link. All the OPM structural links are edges in the $StructLinks$ set and their *type* values correspond to the OPM structural link type.

The OPM graph G is a tuple $\langle V, E \rangle$, where $V = Objs \cup Procs \cup States$ is the set of nodes, and $E = StructLinks \cup ProcLinks$ is the set of directed edges.

Below is a list of additional notations used in the definition of the OPM semantics. The subscript N refers to either a source or a destination node of an edge. For example, instead of defining separately $card_S$ and $card_D$ for source and destination cardinality of an edge, we define below only $card_N$ as a shorthand for both.

- *string* denotes a (possibly empty) sequence of characters.
- The $BasicTypes$ set holds all the OPM built-in types.

$$BasicTypes \doteq \{Boolean, Char, Integer, Float, Double, String,$$

$$Time, Date\} .$$

An OPM object may have one of the OPM built-in types, and the function $type(obj)$ denotes the type of object obj.

- $card_N(l)$ denotes the cardinality of a structural link l on its Source/ Destination and is identical with the corresponding cardinality in the OPM model:

$$card_N : StructLinks \mapsto \{k, k \ldots, k..j, 0 \ldots | k, j \in \mathbb{Z} \wedge k < j\}$$

, where "..." denotes the open range.

In Figure 4.1, all the possible OPM cardinalities are exemplified using unfolding diagram of **Person**. Any **Person** holds an Exhibition-Characterization structural relation with zero to many **Address**es (many is symbolized by "m"), a single **ID**, and one or more **Email**s (symbolized by "+"). In addition, **Person** optionally participates in **Forum** (symbolized by "?") and belongs to one, two or three **Department**s (in the diagram symbolized with a range of two integers "1..3").

In the example, $card_D(Person, Forum) = [0..1]$, $card_D(Person, Email) = [1 \ldots], card_D(Person, Address) = [0 \ldots]$, and $card_D(Person, Department) = [1..3]$.

- $role_N(e)$ denotes the value of $role$ of edge e on its Source/ Destination

$$role_N : E \mapsto string .$$

- $role(v)$ denotes the set of $role$s of a node v

$$role : V \mapsto \text{set of } strings .$$

Both graph edges and graph nodes possess sets of roles defined by the

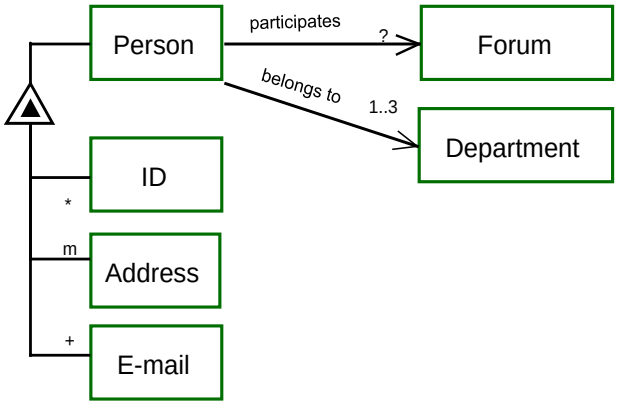


Figure 4.1: Person Properties - OPM model illustrating the OPM cardinalities.

role function. The role sets correspond to the roles of the links and entities in the OPM model. For example, in Figure 4.1, $role_D(Person, Forum) = \{participates\}$.

The roles were added for future extension of the system as they can have important semantic interpretations in advanced OPM mechanisms, but in the current OPM CTS they are not used.

- $type(l)$ denotes the type of a procedural link l.

$$type : ProcLinks \mapsto \{in\text{-}zoomed, unfolded, invocation, instrument,$$

$$condition, consumption, result, effect, minTimeout,$$

$$maxTimeout\} .$$

The $type : ProcLinks$ function, defined over the set of link instances, denotes the type of the OPM link, which the edge represents. Two special cases are (1) the input-output pair in which a process changes an object from its input state to its output state, and (2) the non refined effect link, which is an abstraction of the input-output links pair. In the first case, the OPM input-output links pair can be bound

by the labels of the links, and the links will be represented in the graph via a pair of links of result and consumption types with *label* equal to the link labels in the OPM model. In the latter case, a single effect link will be created.

In Figure 4.2, both effect and input-output link pair are illustrated. The **Marital Status Changing** process has an effect link to the **Marital Status** of **Person**, denoting that the object state or value is changed and the timer associated with the object that calculates the continuous object staying in the same state is reset upon activation of the link. The timer is used to support the object state timeout events. In contrast, the two processes that specialize **Marital Status Changing** refine the link with input-output link pairs. **Divorcing** changes the **Marital Status** from **married** to **divorced** and **Marrying** changes the **Marital Status** from **single** or **divorced** state to **married**.

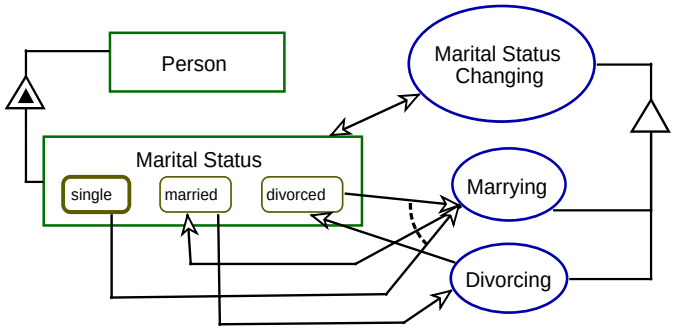


Figure 4.2: OPM Input-output pair of links and effect link examples.

- $function(l)$ denotes the function attached to the procedural link l

$$function : ProcLinks \mapsto \epsilon \cup f(Time, \ldots), \text{ where } \text{“}\ldots\text{”}$$

denotes an unlimited number of arguments. All the arguments except for the first one are of $BasicType$ or of $Objs$ nodes with type belonging to the $BasicType$ set, and $Time$ is defined over $\mathbb{R}$.

- $type(l)$ denotes the type of a structural link l:

  $$type : StructLinks \mapsto \{characterization, specialization, aggregation,$$

  $$association, instantiation\} .$$

- $partialOrder(p)$ is a parameter defined for a process p that has in-zoomed parent process:

  $$partialOrder : Procs \rightarrow PartialOrder ,$$

  $$\text{and } \exists l.\ l \in ProcLinks \wedge type(l) = \text{“}in\text{-}zoomed\text{”} ,$$

  where $PartialOrder$ is a set of natural values. Hierarchical relations among different OPDs is translated into a flattened structure in the OPM graph using procedural links of “in-zoomed” and “unfolded” types between the main OPD thing (from which the refinement started) and the unfolded subprocesses and/or objects. For in-zoomed subprocesses, we keep a $partialOrder$, an integer value which denotes for each process the partial execution order of the process in the normal flow of its parent process.

- $index(o/p)$ denotes the unique identifier of an object o or a process p:

  $$index : Procs \cup Objs \mapsto \mathbb{Z} .$$

  In Figure 3.17, **Lock Acquiring** is the first process to be executed among the subprocesses belonging to the **Milk Supplying** process. Then, **Milk Checking**, **Milk Buying**, and finally **Lock Releasing** subprocesses are executed in this order. Hence, $index(LockAcquiring) = 1$, $index\ (Milk\ Checking) = 2$, $index(MilkBuying) = 3$, and $index\ (LockReleasing) = 4$.

- $minDur(p)$ denotes the minimal duration of a process p:

  $$minDur : Procs \mapsto \mathbb{R} .$$

- $maxDur(p)$ denotes the maximal duration of a process p:

$$maxDur : Procs \mapsto \mathbb{R} \ .$$

- $minDur(st)$ denotes the minimal duration of staying at a state st:

$$minDur : States \mapsto \mathbb{R} \ .$$

- $maxDur(st)$ denotes the maximal duration of staying at a state st:

$$maxDur : States \mapsto \mathbb{R} \ .$$

  The functions $minDur$ and $maxDur$ in the OPM graph denote respectively the minimal and maximal time durations. They are needed for the OPM exception mechanisms and enable definition of the minimal and maximal durations for an object being at a certain state or for a process duration.

- $\varphi_{condition}(p)$ denotes the Boolean precondition on a process p over its incoming edges. Once the process is triggered, it becomes activated if and only if the condition is *true*. Otherwise, the process is *skipped*.

$$\varphi_{condition} : Procs \mapsto LogicalExpr \ ,$$

  where $LogicalExpr$ is defined recursively as follows:

$$LogicalExpr \doteq LogicalExpr \wedge LogicalExpr \mid LogicalExpr$$

$$\vee LogicalExpr \mid \neg LogicalExpr \mid (LogicalExpr)$$

$$\mid PathIdentifier.isActive \ .$$

  $PathIdentifier$ denotes the unique name of an object or a state connected via a procedural link with the process p. In addition, we define for any OPM graph node a Boolean attribute $isActive$, which is true if an object to which the link is connected exists (i.e, there is at least one instance of that object) or the state to which the link is connected is currently the state at which the object is.

  In Figure 4.3, **Registering** to the **Physics 2 Course** is described. The

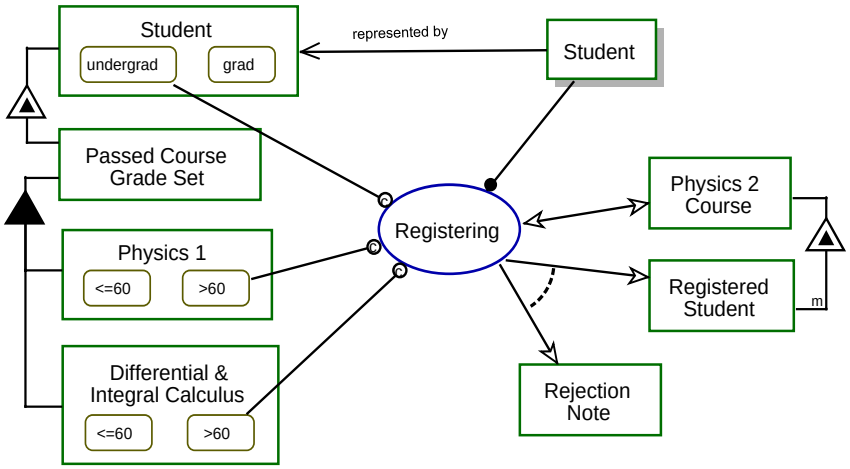


Figure 4.3: Example of condition guards.

precondition of the **Registering** process is that **Student** is at the state **undergrad**. If the condition is not satisfied, the process is skipped. The precondition for this process is expressed in the following equation:

$$\varphi_{condition}(Registering) = [state(Student) = \text{“}undergrad\text{”}] \ .$$

- $\varphi_{wait\_until}(p)$ denotes the Boolean condition of a process p over its incoming edges. Once the process is triggered, it waits until the condition becomes *true*, and then the process is activated.

$$\varphi_{wait\_until} : Procs \mapsto LogicalExpr \ .$$

In Figure 4.3, the process has a wait-until requirement, which means that the process will be waiting until **Student**'s grade for **Physics 1** and for **Differential and Integral Calculus**(D.I.C.) will be in the states more than 60. In addition, a **Physics 2 Course** object instance must exist in order to activate the **Registering** process. The wait_until condition is expressed here in the following equation:

$$\varphi_{wait\_until}(Registering) = (state(Physics\_1) = \text{“} > 60\text{”} \wedge state(D.I.C.) = \text{“} > 60\text{”} \wedge Physics\_2\_Course) \ .$$

- $\varphi_{exits}(p)$ denotes the Boolean condition of a process p over its outgoing edges. Once the process is terminated, the set of activated outgoing links must satisfy the exit condition.

  $$\varphi_{exits} : Procs \mapsto LogicalExpr \ .$$

  By the process termination, the process outgoing links must be selected. Some of the outgoing links are taken due to the OPM execution rules while the process activation, while others must be selected by the process termination. The selected outgoing links must complement the set of outgoing links that were already taken such that the set of all the links taken will satisfy the OPM process exit condition. This condition typically relies on the process entry links. For example, the simplest form of such exit condition is to take the output link in the input-output pair if the corresponding input link is present in the process entry links set.

  In Figure 4.3, the exit condition requires that only one of the result links emanating from **Registering** will be fired. The process generates either a new **Registered Student** or a **Rejection Note**. Formally,

  $\varphi_{exits}(Registering) = (Registered\_Student \times Rejection\_Note)$.

- $prop(o/p)$ denotes the property of an object o or a process p, in the form of an ordered pair (Affiliation, Essence):

  $$prop : Objs \cup Procs \mapsto \{enviromntal, systemic\} \times \{physical,$$

  $$informatical\} \ .$$

  The pair defines the nature of the process/object. The thing can be environmental or systemic and physical or informatical. These attributes have semantic meaning in the CTS, thus they are translated into the OPM graph.

  In Figure 4.3, there are two different OPM objects with the name "S-tudent", the physical one stands for the human student, whereas the second one is the informational object representing the physical student in the OPM model. Both objects are systemic. ting the physical student in the OPM model. Both objects are systemic.

- $ancestors(o/p)$ denotes the set of all the unfolded/in-zoomed ancestors of an object o or a process p:

$$ancestors : Objs \cup Procs \mapsto 2^{Objs \cup Procs} .$$

- $type(o)$ denotes the type of an object o, which can be one of the basic types, an object, or a user-defined aggregation of objects:

$$type : Objs \mapsto T ,$$

where T is defined recursively as follows:

$$T \doteq BasicType \cup Object \cup \{Set < A >, Array < A >, Map < A, B >$$

$$\mid A, B \in T\} .$$

- $domain(o, s)$ defines the range of legal values of an instance of object o, when it is in state s:

$$domain : Objs \times States \mapsto \{(b1, b2), [b1, b2), [b1, b2], (b1, b2] \mid b1, b2 \in$$

$$BasicType\} .$$

- $states(o)$ denotes the set of all the legal states that belong to an object o:

$$states : Objs \mapsto 2^{States} .$$

All the state nodes are connected to the objects to which they belong via the *states* relation. In Figure 4.3, **Student** has two possible states: **undergrad** and **grad**. Formally,

$states(Student) = \{\text{“}undergrad\text{”}, \text{“}grad\text{”}\}$.

- $initSates(o)$ denotes the set of all the initial states that belong to object o:

$$initSates : Objs \mapsto 2^{States} .$$

A set of the initial states of objects is defined by the *initStates* relation. For any object node obj_i, the set $initStates(obj_i)$ must be a proper subset of the set $states(obj_i)$. Note that $initStates(o) \subseteq states(o)$.

- $initVal(o)$ denotes an initial value of object o for which $type(o) \in BasicTypes$, and is undefined for other types of objects:

$$initVal : Objs \mapsto A, \text{ where } A \in BaisicType \ .$$

- $isEventLink(l)$ a Boolean expression which is true if the link l triggers the destination process (i.e., it is an event-link):

$$isEventLink : ProcLinks \mapsto \{true, false\} \ .$$

  The consumption-event and instrument-event links are respectively translated into links of the consumption and instrument types, and in addition, for these links, the $isEvent$ function returns $true$.

- $isBasicProc(p)$ is true if the process p has no in-zoomed or unfolded subprocesses:

$$(\neg \exists l.\ l \in ProcLinks \wedge l.src = proc_i \wedge type(l) \in \{in\text{-}zoomed, unfolded\} \ .$$

- $label(e)$ denotes a label on an edge e. This label is used for binding incoming and outgoing edges connecting a process. Multiple edges can have the same label, defining an execution path.

$$label_N : E \mapsto string \ .$$

- $linksSet_N(p, k)$ denotes all the links connected to a process p tagged with a label k

$$linksSet_N : Procs \times string \mapsto 2^{ProcLinks} \ .$$

OPM Class-Level Graph G^*: We define G^* as the projection of the OPM Graph G, where all the nodes representing object instances and the edges to these instances were removed. Formally:

$$G^* \doteq G - \{obj | obj \in Objs \wedge \exists l.\ l \in StructLinks$$

$$\wedge l.dest = obj \wedge type(l) = instantiation\} \ .$$

4.2 OPM Graph Augmentating

We perform preprocessing, such as adding new links to the OPM graph that are used to preserve process context, during the OPM graph construction. The preprocessing is carried out before generating of the CTS transition rules. The new links have no graphical representation in the OPM model. Following are the graph augmenting rules:

a) Context Preserving Links: this completion of the OPM graph simplifies the creation of characterization-exhibition and aggregation-participation links between newly created objects by some process $proc_m$ and objects in the $proc_m$ context received from its process ancestors. If a subprocess $proc_m$ results or updates or is enabled by an object obj_i, which is an exhibitor or an aggregator of another object obj_j, the parent of $proc_m$ in the OPM graph will hold update or instrument link not only to the object ancestor but also to the object obj_j itself.

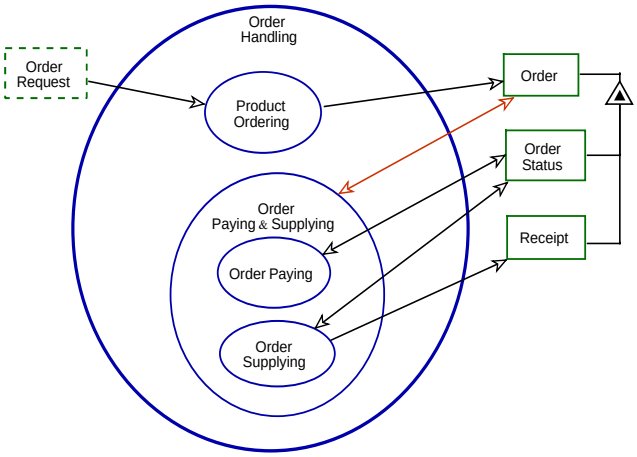


Figure 4.4: Order Handling in-zoomed.

In Figure 4.4, the effect link between **Order** and **Order Paying & Supplying** process was added. The link helps to preserve the **Order** instance created by the **Product Ordering** as part of the **Order Paying & Supplying** process context. Then, **Order Paying**, a subprocess of the **Order Paying & Supplying**, will affect **Order Status**. The **Order Satus** instance that will be selected to be affected will

be the one that is connected via an exhibition-characterization relation with the same **Order** instance that belongs to the **Order Paying & Supplying** process context. Similarly, the **Receipt** instance resulting from **Order Supplying** will be connected via an exhibition-characterization relation to the same **Order** instance. This way, all the subprocesses of **Order Handling** process instance operate on the same **Order** object instance. We call to this manner of object instances selection "context preserving".

b) In-zoomed Process Initialization: a subprocess **Init** is added at the top of each in-zoomed process with $partialOrder(Init) = 0$. The added subprocess simplifies the transition rules related to process invocations in the CTS model. This subprocess also creates all the local objects of the in-zoomed process. In Figure 4.5, **Init** process is added to the **Order Handling** in-zooming diagram. In the example we assume that **Order Supplying** process is invoked up to two times within a single **Order Handling** process run. **Number of Products** is a local object that belongs to the **Order Handling** process. The object is used to calculate number of products that were ordered within one **Order Handling** run. The **Init** process initializes the object.

We continue with basic concepts of OPM-CTS that are important for understanding the operational semantics of OPM. Some of these concepts are based on the definitions in [16]. We conclude this section by defining the abstract CTS execution cycle.

4.3 OPM-CTS Operational Semantics Basic Concepts

We build a CTS $\Phi_{OPM} : \langle Vars, \Theta, \Upsilon, \Pi \rangle$ corresponding to an OPM model, where $Vars$ is a set of variables, Θ is the initial condition, Υ is the transition function, and Π is the time progress condition.

We define $Vars \doteq D \cup C \cup I$, where D is a set of state variables, C is a set of clock variables, including MC – the master clock – and I is a set of Boolean input variables. One of the variables in D is called $SysState$, and it is one of $\{init, tick, event, stepI, stepII, end\}$. Events are a convenient notion for describing some of OPM's operational semantics.

We define the set Ev to hold all the currently unprocessed *events*. We use event to refer to a point in time such as timeout, termination of a process, and creation of a new object. We distinguish between *external* events, which

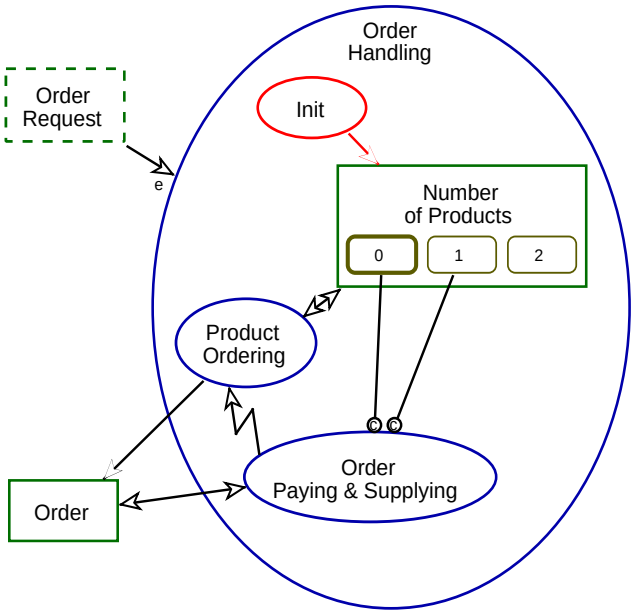


Figure 4.5: Example of Init process added following in-zoomed process initializating rule.

are simply inputs, and *internal* events, which have no inputs associated with them. Technically, each of these event variables (either internal or external) is defined by a specific assignment to state variables, or by a valuation of a predicate (which typically refers to a clock).

In Figure 4.4, an environmental object **Order Request** is associated with an input variable in CTS, which randomly becomes true, whereas an object **Receipt** in the example is created as a result of termination of the systemic process **Order Supplying**. Although the two objects are created following different triggers (internal vs. external event), the transitions that are invoked to create the new object instances are the same. Thus, in both cases we use a Boolean variable *ev_create_object* to represent the event of object creation. This event becomes true and is added to the set Ev of unprocessed events if an object-creating trigger has occurred. During the *stepI* and *stepII* transitions, Ev is used to determine the actions that should be taken, such as creation of a new object instance in the example.

A finite *run* of Φ_{OPM} is a finite sequence of configurations $\sigma_0, \sigma_1, \ldots, \sigma_n$ and corresponding event sets $\{Ev_1, Ev_2, \ldots, Ev_n\}$, where σ_0 is a valid initial configuration, and there exists a valid system step for every two consecutive configurations σ_i, σ_{i+1} for $i \in [0..n-1]$, and a set of unprocessed events Ev_{i+1}. Ev_n contains a *termination* event and $\sigma_n(SysState) = end$.

There are two basic Φ_{OPM} features:

- **Handling of events** – Φ_{OPM} has a clock-asynchronous timing policy, which means that within a single cycle all the inputs, and internal events generated by the system transitions are processed. The system accepts a set of events at the beginning of each *tick*. It takes two *step*s during which it is unstable. Neither the generation of events nor their effect takes time. A process can take longer than a single tick.

- **Execution cycle** – Figure 4.6 shows the OPM execution cycle including all the transitions of the system and their valid sequences. The *init* transition is used to set up the initial configuration $\sigma_0 \in \Theta$. External and temporal events are accepted right after the *tick* transition via the *event* transition, and then, if the event set Ev is not empty, the system becomes unstable, entering a *stepI* transition. New internal events can be generated through this *stepI* transition, leaving the system unstable. Then, the system makes a *stepII* transition at the end of which, the system becomes stable and a new *tick* occurs, beginning the next system cycle.

4.4 A CTS Compliant OPM Model

In the following CTS, we assume that the maximal number of possible object and process instances in the OPM model is bounded by some integer K. Thus, a finite number of variables is sufficient for representing the OPM model using CTS. While we use compound data structures to represent instance states, these data structures can be translated into a flat CTS model (with a larger but still finite number of variables).

Following is a detailed description of all the components of Φ_{OPM}.

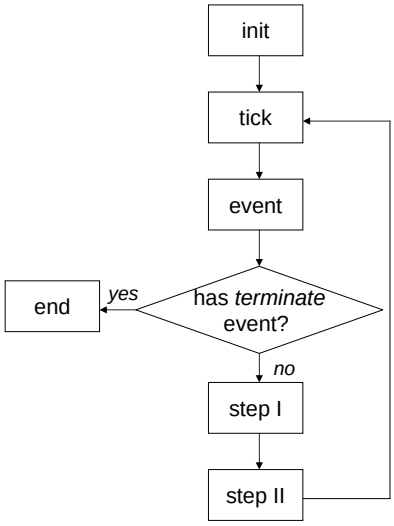


Figure 4.6: The CTS/OPM Execution Cycle

4.4.1 State Variables

Given an OPM class-level graph G^*, let $O = \{o_{ik}\}_{i=1..|Objs|,k=1..K}$ be the set of state variables representing the K instances of each one of the $|Objs|$ objects, and $P = \{p_{jk}\}_{j=T..T+|Procs|,k=1..K}$, where $T = |Objs| + 1$ is the set of state variables representing the K instances of each one of the $|Procs|$ processes in this graph.

Let $Things$ denote the set of all the object and process nodes in the OPM class-level graph G^*. Let also obj_i or $proc_i$ respectively denote an object and a process class node in the OPM graph G^*. For better readability, from now and on we use t_{ij} to denote either o_{ik} or p_{ik}, and $thing_i$ to denote either obj_i or $proc_i$.

Recall that $O \cup P \cup SysState \subset D$, that is a set of CTS state variables. For each link in the OPM class-level graph G^* between node $thing_i$ and $thing_j$, D also includes $K \times K$ Boolean variables, indicating whether each pair of instances in the instance lists of t_{ik_1} and t_{jk_2} are linked in the current system state (configuration).

Following is the complete list of state variable types:

- **Objects.** The structure of a state variable which is an object instance o_{ik}:

  - *val* denotes a pair $< value, state >$ of an object instance o_{ik}, where for the current configuration σ, *value* is the value of o_{ik} and *state* is the state associated with this value. Each element of this pair can be undefined, ϵ. If both *value* and *state* attributes of o_{ik} equal ϵ in the current configuration, the object instance is unused, in such case we will say that $o_{ik} = \epsilon$.

  The creation of an object instance o_{ik} following a result link from some instance p_{jk_1} is possible only if the attribute $o_{ik}.val$ equals ϵ in the current configuration.

- **Processes.** We define a data structure for each process instance p_{ik} that has the following Boolean attribute (fields):

  - *isActive* - *true* if and only if p_{ik} is currently active. Whenever the value of this field becomes "*true*", the process instance is "*activated*".
  - $isWaiting$ - *true* if and only if p_{ik} waits for some precondition to be satisfied. An "*invoked*" process enters a waiting state, that is, its *isActive* field is $false$ and its $isWaiting$ field is *true*.
  - *entries* denotes the set of all the active links that enter into p_{ik}. For example, if for some process p_{ik} $result_{ik,jk_1}$ is true, in the current configuration, then link $l = (proc_i, obj_j)$ from the set E belongs to the entries set of the process p_{ik} in the current configuration.
  - *exits* - denotes the set of all the active links that exit from p_{ik}. For example, if for some process p_{ik}, $consumption_{ik,jk_1}$ is true in the current configuration, then the corresponding link $l = (proc_i, obj_j)$ from the set E belongs to the exit set of the process p_{ik} in the current configuration.
  - $executionSeqNum$ - holds the current timeline of p_{ik}'s execution, that is the partial order of subprocesses that belong to the process that are currently waiting for some precondition or running.

– *subProcs* - denotes the set of all the subprocess instances of the process instance p_{ik}. Formally:
$p_{ik}.subProcs = \{p_{jk_1} \mid unfolding_{ik,jk_1}$
$\vee in\_zooming_{ik,jk_1}\}_{i,j\in[1..|Procs|],k,k_1\in[1..K]}$.

– *ancestors* - denotes the set of all the processes and objects that are the ancestors of the process instance p_{ik}. Formally:
$p_{ik}.ancestors = \{p_{jk_1} \mid p_{ik} \in p_{jk_1}.subProcs$
$\vee \exists p_{hk_2}.\ p_{ik} \in p_{hk_2}.subProcs$
$\wedge p_{jk_1} \in p_{hk_2}.ancestors\}_{i,j,h\in[1..|Procs|],k_1,k_2\in[1..K]}$.

- **Links.** For each procedural or structural link $e \in E$, we define a set of Boolean variables $\{linkType(ik, jk_1)\}_{i,j\in[1..(|Things|]),k,k_1\in[1..K]}$, where $e = (thing_i, thing_j)$ and $type(e) = linkType$ in G^*. Whenever a link instance changes its state such that its value becomes "*true*", we say that the link is "activated" or "fired".

4.4.2 Events

Events are represented by Boolean variables. If the variable becomes *true* (or is "*generated*") in a given configuration then the event happens. For each type of event we list its name, when the event variable is applicable and therefore needs to be added to the system (e.g., an environmental object creation event is not applicable for systemic objects), and when the event becomes *true*. Some of the events are illustrated in the sequel by example diagrams. For each event ev, the indices i and j refer to the first index of an object or a process variable (a thing variable), and the indices k, k_1, and k_2 to refer to its second index. Formally:

$$i, j \in [1..|Things|]; k, k_1, k_2 \in [1..K] .$$

The name of an event variable always has the "*ev*" prefix, followed by underscore, the event type, and identifiers of the relevant object and process variables or just their indeces, if both object and process variables are possible. If the event involves just an object or just a process, the identifier will be o_{ik} or p_{ik}. For example, the name of an event that represents the creation of an environmental instance k of object i is $ev\_create\_o_{ik}$. If the event involves both object and process instances, the identifiers of both things appear in the

event name. For example, the name of the event representing the creation of an object instance o_{ik} by a process instance p_{jk_1} is $ev\_create\_o_{ik}\_p_{jk_1}$. Following are the possible event types and their attributes.

- *System Termination Event*

  Name: $ev\_terminate$

  Exists in D always.

  True if and only if an event of the system termination has occurred.

- *Thing Termination Event*

  Name: ev_term_{ik}.

  Exists in D for each object and process variable.

  True if and only if an event of a process or an object termination has occurred.

The following two event types refer to timeout exception links of a process or an object state. Examples of exception links are given in Figure 4.7. In the diagram, timeout exception links emanate from obj_i and from $proc_j$. Events related to the minimal and maximal exception links will be added to the OPM CTS model for all the object and process variables representing instances of the object and the process. The events are minimal timeout exception event variables for $\{o_{ik}\}_{k=1..K}$ and for $\{p_{jk}\}_{k=1..K}$ and maximal timeout exception event variables for $\{p_{jk}\}_{k=1..K}$.

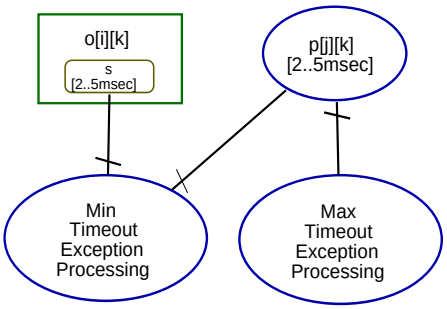


Figure 4.7: OPM Timeout Exception Links

- *Minimal Timeout Exception Event*

  Name: $ev\_min\_timeout_{ik}$.

  Exists in D (1) for a process having a finite minimal duration and at least one outgoing timeout event link related to this constraint, and (2) for an object having at least one state with minimal staying time constraint that is connected with at least one timeout event link. Formally:

  $$ev\_min\_timeout_{ik} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge type(l) = \text{“}minTimeout\text{”} \wedge (l.src = proc_i \vee (l.src = s \wedge s \in states(obj_i))) \ .$$

  True if and only if a timer $time_{ik}$ related to the instance of an active process p_{ik} or an existing object o_{ik} has reached its minimal boundary.

- *Maximal Timeout Exception Event*

  Name: $ev\_max\_timeout_{ik}$.

  Exists in D for a process having a finite maximal duration and at least one outgoing timeout event link related to the constraint. In addition, the event is defined for an object having at least one state with maximal staying constraint which is connected with at least one timeout event link related to the constraint. Formally:

  $$ev\_max\_timeout_{ik} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge type(l) = \text{“}maxTimeout\text{”} \wedge (l.src = proc_i \vee (l.src = s \wedge s \in states(obj_i))) \ .$$

  True if and only if a timer $time_{ik}$ related to the instance of an active process p_{ik} or an existing object o_{ik} has reached its maximal value.

- *Environmental Object Creation Event*

  Name: $ev\_create\_o_{ik[s]}$.

  Exists in D for an environmental object that is not a component or an attribute of any other thing (object or process). The event name optionally specifies the initial state s of the created object instance.

Formally:

$ev\_create\_o_{ik[s]} \in D \leftrightarrow prop(obj_i)[affiliation] = \text{“}environmental\text{”}[\wedge$
$s \in initStates(obj_i)] \wedge (\neg \exists l.\, l \in ProcLinks \wedge l.dest = obj_i \wedge$
$type(l) \in \{characterization\,, aggregation\,, instantiation\})\ .$

True if and only if an external event of an object creation occurred during an event transition.

- *Object Creation Event*

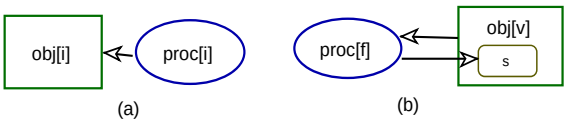


Figure 4.8: OPM Object Creation and Effect

Name: $ev\_create\_o_{ik_1[s]}\_p_{jk_2}$

Exists in D for object that is connected with a process via a result link. Formally:

$ev\_create\_o_{ik_1[s]}\_p_{jk_2} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge$
$(l = (obj_i, proc_j) \vee (l = (s, proc_j) \wedge s \in states(obj_i))) \wedge$
$type(l) = \text{“}result\text{”}\ .$

In Figure 4.8(a), $proc_j$ yields obj_i, so a set of events

$\{ev\_create\_o_{ik}\_p_{jk_1}\}_{k,k_1 \in [1..K]}$ will be added to the model.

True if and only if one of the processes connected with the object via a result link terminates such that the link is activated.

- *Object Effect Event*

Name: $ev\_modify\_o_{ik_1[s]}\_p_{jk_2}$.

Exists in D for an object that is connected with a process via an effect

link. Formally:

$$ev\_modify\_o_{ik_1[s]}\_p_{jk_2} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge$$
$$(l = (obj_i, proc_j) \vee (l = (s, proc_j) \wedge s \in states(obj_i))) \wedge$$
$$type(l) = \text{“}effect\text{”}\ .$$

True if and only if a process connected with the object via the effect link terminates such that the link is activated.

In Figure 4.8(b), $proc_f$ transforms obj_v from any state into the state s, so a set of events $\{ev\_modify\_o_{vks}\_p_{fk_1}\}_{k,k_1 \in [1..K]}$ is added to the model.

4.4.2.1 Control-Flow Related Events

There are three control flow related events: a process invocation event, a process iteration event and an event of change in the in-zoomed process execution order. Figure 4.9 exemplifies each one of these events.

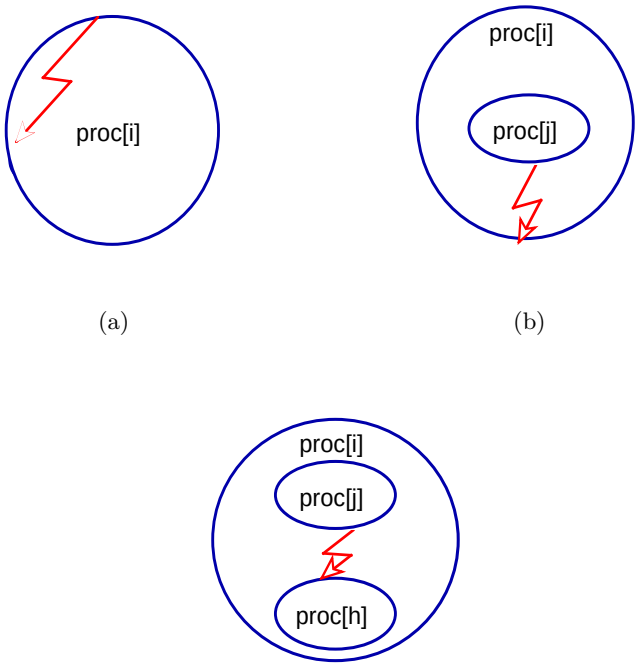
When a process invocation event occurs, the context in which the process is invoked is “inherited” from the thing that has invoked the process, the input set or/and components and features of the parent is used by its subprocesses. All the local objects of the process are created upon the process actual activation.

If the process iterates, the context of the process already exists and it is reused. Thus, all the local objects of the process instance preserve their state from the previous iteration. Handling of an event of a change in the process execution flow is similar to handling of a process iteration event in the sense that the process context is preserved, all the currently running subprocess instances related to the process are halted, and the $processExecutionSeqNum$ attribute of the process instance is changed in accordance with the fired invocation or event link.

In the case of an event of a change in the process normal flow, some processes are “halted” – stopped in the middle of their execution, their $isActive$ and $IsWaiting$ fields become $false$, and they do not result/modify any objects during the halt.

- *Process Invocation Event*

  Name: $ev\_invoke\_p_{ik_1}\_t_{[jk_2[s]]}$.



(a) (b) (c)

Figure 4.9: Control flow related events: (a) Self-process invocation, (b) Process iteration - invokes the next iteration of the parent process, and (c) Change in the control flow of a process $proc[i]$ by its subprocess $proc[j]$.

Exists in D for a process with at least one incoming invocation or event link. The first identifier denotes in the event p_{ik}, the process that is to be invoked, and the second identifier refers to the invocation link source, which can be an object o_{jk_1}, or its state s or another process p_{jk_1}. Formally:

$$ev\_invoke\_p_{ik\text{-}jk_1[s]} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge (l = (proc_i, proc_j) \vee l = (proc_i, obj_j) \vee (l = (proc_i, s) \wedge s \in states(obj_j))) \wedge isEventLink(l)\ .$$

True if and only if the link $invocation_{ik,jk_1[s]}$ is activated as a result of the source process termination or as a result of the source object creation or the object entering its state s. Self invocation is also possible.

$$ev\_invoke\_p_{ik_1} \in D \leftrightarrow prop(proc_i)[affiliation] = \text{“}environmental\text{”} \wedge (\neg\exists l.\, l \in ProcLinks \wedge l.dest = proc_i \wedge type(l) \in \{characterization, aggregation\}) \ .$$

- *Process Iteration Event*

  Name: $ev\_iterate\_p_{ik}$.

  Exists in D for process having an in-zooming diagram. Formally:

  $$ev\_iterate\_p_{ik} \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge l.src = proc_i \wedge type(l) = \text{“}in\text{-}zoomed\text{”} \ .$$

  True if and only if a new iteration of the currently active process occurs (meaning the same context, i.e., the same inputs object set).

- *Process Execution Flow Event*

  Name: $ev\_change\_executionOrder_1\_p_{ik}\_executionOrder_2$.

  Exists in D for a process having in-zooming diagram. Formally:

  $$ev\_change\_executionOrder_1\_p_{ik}\_executionOrder_2 \in D \leftrightarrow \exists l.\, l \in ProcLinks \wedge l.src = proc_i \wedge type(l) = \text{“}in\text{-}zoomed\text{”} \wedge executionOrder_1, executionOrder_2 \in PartialOrder \ .$$

  The event denotes modifying the process current timeline, where

  $executionOrder_2$ stands for the new timeline of the process and

  $executionOrder_1$ is the previous one.

  True if and only if an invocation or an event link to one or more of the subprocesses belonging to the process p_{ik} was fired.

4.4.3 Clock Variables

The following elements comprise the set C of the clock variables.

- $time_{ik}$ denotes an object or a process timer. It is defined for each thing and keeps the time from the last modification of an object instance or from an activation of a process instance. A clock variable exists for each object having a state which is a source of an exception link and for each process which is a source of a process timeout exception link. If the process instance is not active or the object instance does not exists, the timer value is undefined.

$$time_{ik} \in C \leftrightarrow i \in [1..|Things|], k \in [1..K] \ .$$

- MC denotes the master clock.

All the timers, including the master clock, are reset at the *init* transition. Any variable, except for the master clock, can be reset through any transition during the execution cycle.

4.4.4 The Initial Configuration

The initial configuration of any Φ_{OPM} computation starts with (1.0)[1] all the clocks, including the master clock MC, reset to zero, (1.1) the next $SysState$ is set to the value "*tick*", and (1.2,1.3) the values of all the object variables O in the next configuration are initialized according to the OPM instance graph G/G^{*2}. Formally:

$\Theta : \forall i, \forall k.\ i \in [1..|Things|] \wedge k \in [1..K] \wedge time_{ik} \in C$

(1.0) $time'_{ik} = 0 \wedge MC' = 0$

(1.1) $SysState =$ "$init$" $\wedge\ SysState' =$ "$tick$"

(1.2) $\forall i, k.\ i \in |1..|Objs|| \wedge k \in [1..K] \wedge obj_i \in G/G^* \wedge \exists l.\ l \in StructLinks \wedge type(l) =$ "$instantiation$" $\wedge\ l = (obj_i, obj_k) \rightarrow o_{ik}.val[0]' = initVal(obj_k) \wedge o_{ik}.val[1]' = initState(obj_k)$

(1.3) $\forall i, k.\ i \in |1..|Objs|| \wedge k \in [1..K] \wedge obj_i \in G/G^* \wedge \neg\exists l.\ l \in StructLinks \wedge type(l) =$ "$instantiation$" $\wedge\ l = (obj_i, obj_k) \rightarrow o_{ik}.val[0]' = \epsilon \wedge o_{ij}.val[1]' = \epsilon$

Figure 4.10 demonstrates an OPM model that has an initial population. The population includes two instances of the object **Eukaryotic Cell**. Assume that the **Eukaryotic Cell** object is represented in the OPM graph by the node

[1]The number refer to the equation numbers.

[2]Nodes that exist in the instance graph G but not in the class-level graph G^*.

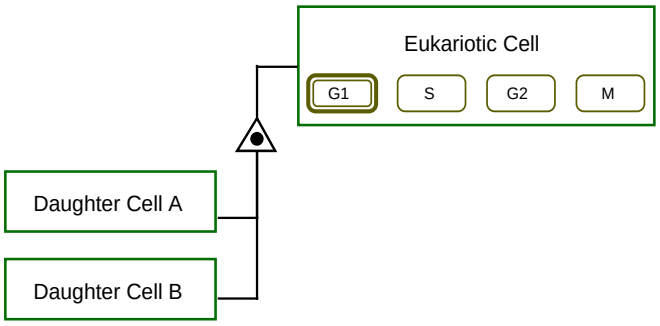


Figure 4.10: Eukaryotic Cell Object with two Instances. In the diagram, **Eukaryotic Cell** instantiates two **Daughter** cells via relation of "instantiation" type.

obj_1, and its two instance objects by the nodes obj_2 and obj_3. Assume also that the objects have time constraints. The following formula represents the initial configuration of the associated CTS model:

$\phi : o_{11}.val' = (\epsilon, \text{“}G1\text{”}), o_{12}.val' = (\epsilon, \text{“}G1\text{”}) \wedge MC = 0 \wedge time_{11} = 0$
$\wedge time_{12} = 0$

4.4.5 Tick Transition

The *tick* transition relation is given by the ρ_{tick} function:

$\rho_{tick} : \quad Ev = \emptyset \wedge SysState = \text{“}tick\text{”} \wedge \exists \Delta.\ \Delta > 0 \wedge SysState' = \text{“}event\text{”}$
$\wedge D'/SysState' = D/SysState \wedge C' = C + \Delta$

Note that the transition precondition implies that there are currently no unprocessed events in the set of events Ev.

4.4.6 Event Transition

In this section we provide the formal definition of the event transition, denoted by τ_{event}. The transition is defined as a conjunction of the following rules.

4.4.6.1 The *event* transition precondition rule

The transition always happens after the *tick* transition. The precondition of the *event* transition is that the event set Ev has no unprocessed events from the previous iteration and the value of the variable $SysState$ equals the "event". Formally:

(2.0.0) $Ev = \oslash \wedge SysState =$ "$event$" .

(2.0.1) $SysState' =$ "$stepI$" .

4.4.6.2 The process termination event $ev\_term\_p_{ik}$ rule

This rule defines the generation of a process termination event,

$ev\_term\_p_{ik}$, for some process p_{ik}. A process termination event is possible in the next configuration only if the following conditions hold:

- the process is active in the current configuration (2.1.1),
- all the subprocesses are neither active nor waiting for some precondition (2.1.2).

When the termination event occurs, we define the process to be inactive and not waiting (2.1.3). Formally:

(2.1.0) $ev\_term\_p_{ik} \in Ev' \rightarrow$	//Termination of a process p_{ik} in the next configuration implies the following:
(2.1.1) $p_{ik}.isActive$	//In the current configuration the process is active.
(2.1.2) $\wedge \neg\exists child.\ child \in p_{ik}.subProcs \wedge (child.isActive \vee child.isWaiting)$	//All its subprocesses are neither active nor waiting.
(2.1.3) $p_{ik}.isActive') \wedge \neg(p_{ik}.isWaiting')$	//Then in the next configuration the process will be inactive and not waiting.

4.4.6.3 Process exit links related events generation rules

As a result of a termination event $ev\_term\_p_{ik}$ occurrence, several exit links from process p_{ik} are activated, and several events related to these links are generated. Let S denote this set of exit links. S must satisfy two conditions: (1) combined with already-activated process exit links, the set must hold the process exit condition $\varphi_{exits}(proc_i)$ and (2) path-compatibility with all the incoming links of the process must be maintained. Formally:

(2.2.0) $\wedge_{s \in S \cup p_{ik}.exits} \varphi_{exits}(proc_i)$	$\models$	//The selected set S of exit links together with the process exit links that have already being activated by the subprocesses of the process p_{ik} satisfy the process $proc_i$ exit condition.
(2.2.1) $\wedge\, S \bigcap p_{ik}.exits = \oslash$		//The exit link instance is not activated twice.
(2.2.2) $\forall l, \forall u.\ l \in p_{ik}.entries \wedge u \in S \cup p_{ik}.exits \wedge (l.src = obj_j \vee (l.src = s_1 \wedge s_1 \in states(obj_j))) \wedge (u.dest = obj_j \vee (u.dest = s_2 \wedge s_2 \in states(obj_j))) \rightarrow label(l) = label(u) \vee label(u) = \text{“ ”} \vee label(l) = \text{“ ”}$		//The incoming link emanating from obj_j or from one of its states, the outgoing link enters the same object obj_j or one of its states, and the input-output link pair has the same label (path-compatibility).

We denote the conjunction of all the listed conditions (2.2.0-2.2.2) by *Scond*. We define a function $fire : 2^{ProcLinks} \times P \rightarrow 2^{Events}$ which, given the set of exit links S, returns the set of events that must now be generated, the $\bar{S}$ set. This function will be described later in this section. To summarize, the relation between the termination event and the events that are created is:

(2.2.3) $ev\_term\_p_{ik} \in Ev' \wedge \exists S.\ S \subseteq ProcLinks \wedge Scond \rightarrow fire(S, p_{ik}) \subseteq Ev'$	//If a process termination event occurs and there exists a set of exit links S, such that $Scond$ is satisfied, then the events related to activation of the links in S are added to Ev'.

To illustrate this rule, let us assume that the process $proc_i$ in Figure 4.11, is to be terminated in the next configuration and that the process inputs include an object obj_h in state $s1$. Then, the event $ev\_term\_p_{ik}$ is in Ev', and according to (2.2.2), S includes links from the process $proc_i$ to $obj_j.s2$ or to $obj_j.s3$, but not the state $obj_j.s4$. This happens because the labels of the links $(obj_j.s1, proc_i)$ and $(proc_i, obj_j.s4)$ are not compatible, because they hold unequal and non-empty values. On the other hand, under our assumptions, an exit link $(proc_i, obj_h.s2)$ is possibly valid because the labels bind only input-output link pairs that connect the same object with the same process nodes.

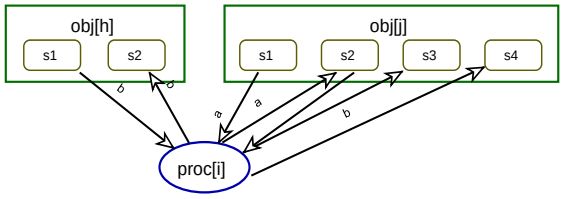


Figure 4.11: Bounding of the incoming and outgoing links via labels.

4.4.6.4 Environmental object creation event rule

A creation event of an environmental object can be generated only if the following conditions hold:

- the object being created is defined as environmental (2.3.0),

- the object is not an attribute or part of any other process or object (2.3.1),
- none of the processes in the model is connected to the object with a result link (2.3.2), and
- if the created object has initial states, the object will be generated in one of the initial states (2.3.3).

All these conditions are checked while creating event variables of this type. The conjunction of the following expressions defines the rule for creating an environmental object o_{ik}:

(2.3.0) $ev\_create\_o_{ik[s]} \in Ev' \rightarrow o_{ik}.prop[affiliation] =$ "*environmental*"	//Creation of an environmental object instance o_{ik} in the next configuration is possible if following conditions are satisfied.
(2.3.1) $\neg\exists l.\ l \in StructLinks \wedge type(l) \in$ "*characterization*", "*aggregation*" $\wedge$ $l.dst = obj_i$	//The object is not target of a characterization or an aggregation structural link.
(2.3.2) $\neg\exists l.\ l \in ProcLinks \wedge type(l) =$ "*result*" $\wedge\, l.dst = obj_i$	//The object cannot be created via a result link.
(2.3.3) $o_{ik}.val[0]' = initVal(obj_i) \wedge [initStates(obj_i) \neq \oslash \rightarrow \exists s.\ s \in initStates(obj_i) \wedge o_{ik}.val[1]' = s]$	//Then the new instance will be created in one of the possible initial states

4.4.6.5 Timeout related events

Timeout related events are generated through the rules (2.4.0 - 2.4.3). Recall that for every timeout exception link in the OPM model, we add a set of event variables in the CTS model. The following rules define their behavior:

(2.4.0) $ev\_max\_timeout\_p_{ik} \in Ev' \leftrightarrow t_{ik} \geq maxDur(proc_i) \wedge p_{ik}.isActive$	//The process timer calculates the duration from the process invocation. If the process was invoked, but due to some unsatisfied *wait_until* preconditions was not activated for some time T, this time is calculated within the process timer t_{ik}.
(2.4.1) $ev\_max\_timeout\_o_{ik} \in Ev' \leftrightarrow o_{ik} \neq \epsilon \wedge \exists l.\ l \in ProcLinks \wedge l.src = s \wedge o_{ik}.val[1] = s \wedge type(l) = \text{“}maxTimeout\text{”} \wedge t_{ik} \geq maxDur(s)$.	//The current state of the objects o_{ik} emanates from at least one timeout exception link.
(2.4.2) $ev\_min\_timeout\_p_{ik} \in Ev' \leftrightarrow t_{ik} \geq minDur(proc_i) \wedge p_{ik}.isActive$.	//A process timeout exception occurs if and only if the process timer reaches the process minimal duration and the process is invoked (but it may still be in the waiting state).
(2.4.3) $ev\_min\_timeout\_o_{ik} \in Ev' \leftrightarrow o_{ik} \neq \epsilon \wedge \exists l.\ l \in ProcLinks \wedge l.src = s \wedge o_{ik}.val[1] = s \wedge type(l) = \text{“}minTimeout\text{”} \wedge t_{ik} \geq minDur(s)$.	//An object timeout exception occurs if and only if there exists a timeout exception link that emanates from the object current state and the object timer reached the state minimal duration.

4.4.6.6 Fire Function

The function $fire : 2^{ProcLinks} \times P \mapsto 2^{Event}$, used in the *event* transition relation is defined as follows.

$fire(S, p_{ik}) = \bar{S}$, where $\bar{S}$ is defined through the following rules (R1) through (R3).

F1: Firing a result link related event

For any link of type "*result*" that emanates from the process $proc_i$ and points at an object obj_j, an object creation event is generated if all the following conditions hold:

- The process $proc_i$ is non-compound (i.e., basic, having no subprocesses) and either it is not part of an input-output links pair, or, if it has subprocesses, none of these subprocesses refines the result link. In Figure 4.12, all the fired result links are colored in red, whereas a result link or output links from the input-output link pair that do not result in creation of a new object are colored in blue. In the example, both $proc_i$ and $proc_j$ are basic, still $proc_j$ won't create any object, since its result link to the object obj_d is part of the input-output link pair. $proc_i$ will generate a new object obj_d upon the process termination.

  In-zoomed processes can also generate new object instances if the result link they hold is not refined by any one of their subprocesses. In the example, $proc_f$ will generate the object obj_g. Actually, the process will assemble subcomponents of obj_g created by its subprocesses. In contrast, $proc_h$ won't generate obj_g because a special assembling subprocess $proc_f$ will do the assembly work.

- None of the process ancestors (at any level above that process) defines an object state or value update using an input-output link pair or an effect link.

Formally:

(3.0.0) $\forall l.\ l \in S \wedge type(l) =$ "*result*" $\wedge\, l = (proc_i, obj_j) \wedge$	//For each result link from $proc_i$ to obj_j
(3.0.1) $((isBasicProc(proc_i) \wedge \neg\exists d.\ d \in ProcLinks \wedge (d.dest = obj_j \vee (d.dest = s \wedge s \in states(obj_j))) \wedge type(d) =$ "*consumption*")	//The process is basic and has no consumption link emanating from the object of from one of its states.

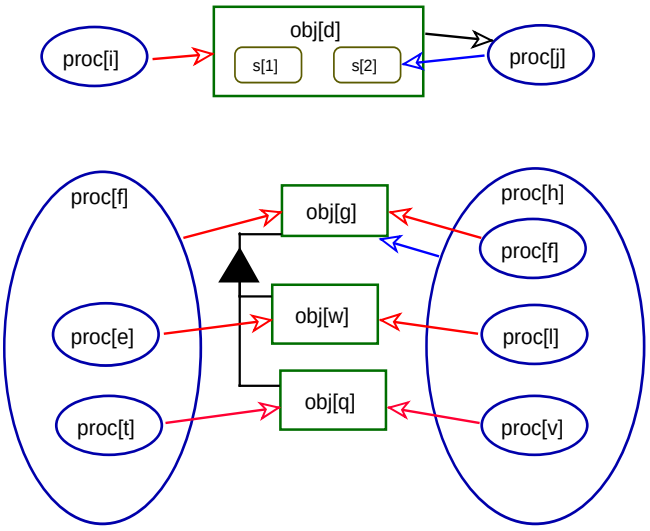


Figure 4.12: OPM Result Links Exemplified.

(3.0.2) $\vee(\forall x.\ x \in ProcLinks \wedge x = (proc_i, proc_h) \wedge type(x) \in \{in\text{-}zoomed, unfolded\} \wedge \neg\exists g.g \in ProcLinks \wedge (g = (proc_h, obj_j) \vee (g = (proc_h, s_1) \wedge s_1 \in states(obj_j))) \wedge type(g) = \text{“}result\text{”})$

//No one of the subprocesses belonging to the process $proc_i$ has a result link with the object obj_j.

(3.0.3) $\neg(\exists h.\ h \in [1..|Proc|] \wedge proc_h \in ancestors(proc_i) \wedge \exists l_1.\ l_1 \in ProcLinks \wedge (l_1 = (proc_h, obj_j) \wedge type(l_1) = \text{“}effect\text{”}) \vee (l_1 = (proc_h, s) \wedge s \in states(obj_j) \wedge type(l_1) = \text{“}consumption\text{”}))).$

//No one of the process ancestors holds an effect or a consumption link with the object.

(3.0.4) $\wedge \exists k_1.\ k_1 \in K \wedge o_{jk_1}.val = \epsilon$ //There exists an unused object identifier.

(3.0.5) $\rightarrow ev\_create\_o_{jk_1}\_p_{ik} \in \bar{S}$. //An object creation event is added to the set $\bar{S}$.

To summarize this rule, for any result link connecting the process $proc_i$ with an object obj_j from equation (3.0.0) that holds conditions (3.0.1) or (3.0.2) and the conjunction of the conditions (3.0.3) to (3.0.4), an event creating the object instance is added to the set $\bar{S}$.

F2: Firing the refined result link related events

This rule is similar to the object creating rule, except that the fired link destination is an object state. Formally:

(3.1.0) $\forall l.\ l \in S \wedge type(l) =$ "$result$" $\wedge\ l = (proc_i, s_1) \wedge s_1 \in states(obj_j)$ //For each result link from $proc_i$ to the state s_1 belonging to the object obj_j

(3.1.2) $((isBasicProc(proc_i) \wedge \neg\exists d.\ d \in ProcLinks \wedge (d.dest = obj_j \vee (d.dest = s \wedge s \in states(obj_j))) \wedge type(d) =$ "$consumption$") //The process $proc_i$ is basic and there is no consumption link d between the process and the object obj_j

(3.1.3) $\vee (\forall l.\ l \in ProcLinks \wedge l = (proc_i, proc_h) \wedge type(l) \in \{in\text{-}zoomed, unfolded\} \wedge \neg\exists g.\ g \in ProcLinks \wedge ((g = (proc_h, s_1) \wedge s_1 \in states(obj_j)) \vee g = (proc_h, obj_j)) \wedge type(g) =$ "$result$") //If the process $proc_i$ has subprocesses, no one of the subprocesses has a result link to the object obj_j.

(3.1.4) $\neg\exists h, k_3.\ p_{hk_3} \in P \wedge (p_{hk_3} \in p_{ik}.ancestors \wedge \exists l_1.\ l_1 \in ProcLinks \wedge\ (l_1 = (proc_h, obj_j) \wedge type(l_1) = \text{“}effect\text{”}) \vee (l_1 = (proc_h, s_2) \wedge s_2 \in states(obj_j) \wedge type(l_1) = \text{“}consumption\text{”})$	//The process $proc_i$ has no ancestors with an effect link or an input-output link pair connecting it with the object obj_j
(3.1.5) $\rightarrow ev\_create\_o_{jk_1s_1}\_p_{ik} \in \bar{S}$.	//If the conditions are satisfied, an event creating an instance of object obj_j at state s_1 is added to the set $\bar{S}$.

F3: Firing the the effect event

An event of an object obj_j effect is generated if the following conditions hold:

- $proc_i$ has an effect link or at least one from input-output link pair (links of "$consumption$" and "$result$" types) defined by its parent which connects the process with the object.

Figure 4.13 demonstrates the possible cases, in which the basic process $proc_i$ will generate an event of modifying an object obj_j upon the process termination.

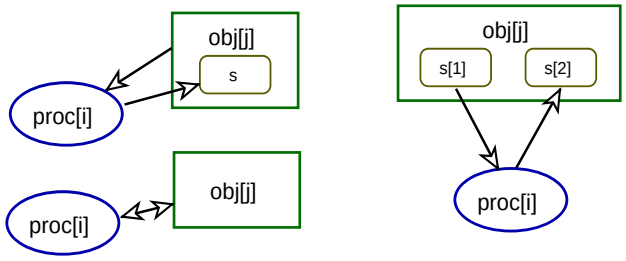


Figure 4.13: OPM Effect Links Exemplified.

(3.2.0) $\forall l.\ l \in S \wedge type(l) =$ "$effect$" $\wedge\, l.dest = obj_j$	//There exists an effect link that targets an object obj_j from the process.
(3.2.1) $\vee\ (type(l) =$ "$result$" $\wedge$ $(l.dest = obj_j \vee (l.dest = s \wedge s \in states(obj_j))) \wedge \exists l_1 .l_1 \in ProcLinks \wedge type(l_1) =$ "$consumption$" $\wedge\ (l_1 = (proc_i, obj_j) \vee (l_1 = (proc_i, s_1) \wedge s_1 \in states(obj_j))))$	//The result link is one of the links constructing input-output pair of links.
(3.2.2) $\exists k_1 .k_1 \in [1..K] \wedge consume_{ik,jk_1} \vee effect_{ik,jk_1}$	//The process entries set includes an object o_{jk_1} that should be modified.
(3.2.3) $isBasicProc(proc_i) \rightarrow (l.dest \in states(obj_j) \rightarrow ev\_modify\_o_{jk_1[s]}\_p_{ik} \in \bar{S}) \vee (l.dest \notin states(obj_j) \wedge$	//If $proc_i$ is non-compound and also the fired link targets a state s, the object will be transformed to this.
$\exists s_2 \in states(obj_j) \rightarrow ev\_modify\_o_{jk_1 s_2}\_p_{ik} \in \bar{S}) \vee (l = (proc_i, obj_j) \wedge initSates(obj_j) = \oslash \wedge ev\_modify\_o_{jk_1}\_p_{jk}) \in \bar{S}.$	//If the link targets a stateful object, one of its states is selected.

F4: Firing the invocation link related events

The following rules generate events associated with firing of invocation links belonging to the exit set S of a process p_{ik}. Any generated event is added to the result set $\bar{S}$ of $fire$.

Case (a) An invocation link from the process $proc_i$ which is a subprocess of $proc_a$ targets one of the subprocesses enclosed by the process in-zoomed

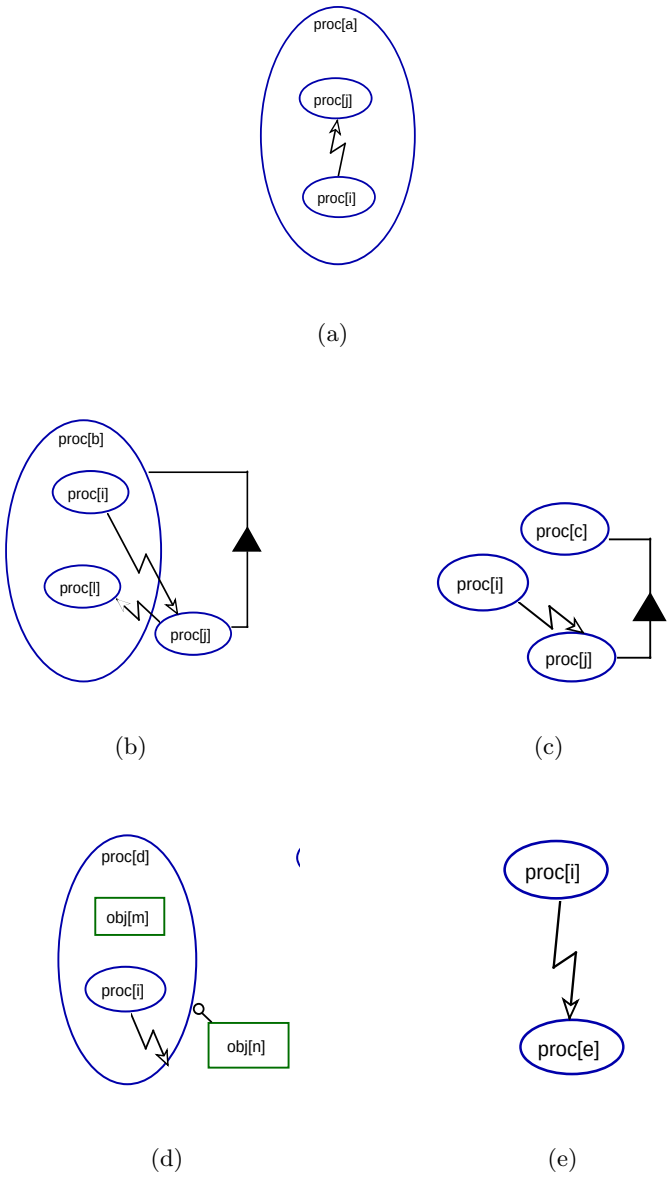


Figure 4.14: OPM invocation links exemplified.

ancestor. In addition, the targeted process itself is not one of $proc_i$ ancestors. In this case, the event of changing the current timeline of the closest common ancestor is generated. In Figure 4.14, two subprocesses of **proc[a]**[1] have a common in-zoomed ancestor. The subprocess **proc[i]** has an invocation link to the subprocess **proc[j]**. If the invocation link is fired according to the rule described below, the timeline of the process **proc[a]** will be changed, all the currently running subprocesses of the process will be stopped, and all the subprocesses in the same timeline of **proc[j]** will be invoked. These subprocesses will be invoked in the context of the currently running instance of **proc[a]**. Formally:

(3.3.0) $\forall l.\ l \in S \wedge l = (proc_i, proc_j) \wedge type(l) =$ "*invocation*"	//The rule is defined for any invocation link l that belongs to the OPM graph.
(3.3.1) $\wedge\ proc_j \notin ancestors(proc_i) \rightarrow$ $ev\_change\_t_{partOrder(proc_j)}\_p_{nk_1}$	//$proc_j$ is not an ancestor of $proc_i$. //An event of change in the execution order of the process $proc_j$ is added to $\bar{S}$.
$\_t_{p_{nk_1}}.execSeqNum \in \bar{S}$	

Case (b) An invocation link from the process $proc_i$ targets a process that is a subprocess of an active process or an attribute of an existing object. In addition, the invoked process is not enclosed by any in-zoomed process that has a common ancestor with the process $proc_i$. In Figure 4.14, a process **proc[b]** has two subprocesses, **proc[i]** and **proc[j]**. The second subprocess is not part of the in-zoomed process, and the two subprocesses have a common ancestor. Firing of the invocation link between **proc[i]** and **proc[j]** will cause stopping of all the currently running **proc[b]** in-zoomed subprocesses and will invoke **proc[j]** in the context of the current instance of the process **proc[b]**. In the example, we show how **proc[j]** can then return the process **proc[b]** into the

[1] **proc[a]** in the OPM diagram denotes $proc_a$.

in-zoomed flow using another invocation link to the subprocess **proc[l]**. This pattern can be used in situations when the first invocation link is invoked upon occurrence of an error and after treating the error, the normal flow is resumed. For example, a program tries to open a file, but currently the file is in use and the file system forbids the action. The user is notified regarding the failure and then he/she closes the file and the program tries opening the file for the second time, this time with success. Finally, the normal flow is resumed. Formally:

(3.4.0) $\forall l.\ l \in S \wedge l = (proc_i, proc_j) \wedge type(l) =$ "*invocation*"	//The rule is defined for any invocation link l that belongs to the OPM graph.
(3.4.1) $\wedge proc_j \notin ancestors(proc_i)$	//$proc_j$ is not an ancestor of $proc_i$
(3.4.2) $\wedge \exists n.\ n \in [1..\|Procs\|] \wedge thing_n \in ancestors(proc_i) \wedge thing_n \in ancestors(proc_j)$	//The destination and target processes have a common ancestor $thing_n$.
(3.4.3) $\wedge \forall h.\ h \in [1..\|Procs\|/\|Objs\|] \wedge h \neq n \wedge thing_h \in ancestors(proc_i) \wedge thing_h \in ancestors(proc_j) \wedge thing_h \in ancestors(thing_n)$	//The ancestor $thing_n$ is the closest common ancestor of the two subprocesses in the refinement tree.
(3.4.4) $\wedge !\exists l_1.\ l_1 \in ProcLinks \wedge l_1 = (thing_n, proc_j) \wedge type(l_1) =$ "*in-zoomed*"	//The target process is enclosed by the process $proc_n$ in-zoomed diagram.
(3.4.5) $\wedge \exists k_1.\ k_1 \in [1..K] \wedge thing_{nk_1} \in p_{ik}.ancestors$	//Find the ancestor instance in the context of which the terminating process p_{ik} runs.

(3.4.6) $\wedge\ \exists k_2 \in [1..K] \wedge \neg p_{jk_2}.isActive \wedge \neg p_{jk_2}.isWaiting$	//The process instance p_{jk_2} is not in use in the current configuration.
(3.4.7) $\rightarrow$ $ev\_invoke\_p_{jk_2}\_thing_{nk_1} \in \bar{S}$	//If all the conditions are satisfied, a new invocation event is added to the returned set, and a new process instance p_{jk_2} is created in the next configuration.

Case (c) An invocation link from the process $proc_i$ targets a process that is a part of a currently active process or characterizes an existing object. In addition, the invoked process is not enclosed by any in-zoomed process and has no common ancestor with the process $proc_i$. In Figure 4.14, a process **proc[c]** has no in-zooming diagram and it has also a subprocess **proc[j]**. A process **proc[i]** has an invocation link to the process **proc[j]**. If **proc[c]** is active while the invocation link is fired, the process **proc[j]** will be invoked in a context of any active **proc[c]** process instance.

(3.5.0) $\forall l.\ l \in S \wedge l = (proc_i, proc_j) \wedge type(l) =$ "*invocation*"	//The rule is defined for any invocation link l that belongs to the OPM graph.
(3.5.1) $\wedge\, proc_j \notin ancestors(proc_i)$	//$proc_j$ is not an ancestor of $proc_i$.
(3.5.6) $\rightarrow$ $ev\_invoke\_p_{jk_2}\_thing_{fk_1} \in \bar{S}$	//If the conditions are satisfied, an invocation event creating a new process instance for $proc_j$ is added to the set $\bar{S}$.

Case (d) The invocation link from the process $proc_i$ targets one of its ancestors. Then, the *ev_iterate* event will be generated. In Figure 4.14, the process **proc[i]** is a subprocess of **proc[d]** and it has an invocation link to its parent. Firing of the link will cause iteration of the process **proc[d]**, its objects input set will stay the same, whereas all the local objects (**obj[m]** in the example) will be re-created in the next iteration.

(3.6.0) $\forall l.\ l \in S \wedge type(l) =$ "*invocation*" $\wedge\, l = (proc_i, proc_j) \wedge$ $proc_j \in ancestors(proc_i) \wedge$ $\exists k_1.\ k_1 \in \lvert 1..K \rvert \wedge p_{jk_1} \in$ $p_{ik}.ancestors$	//The invoked process $proc_j$ is an ancestor of $proc_i$, and there is an ancestor relation between the process instances p_{jk_1} and p_{ik}.
(3.6.1) $\rightarrow ev\_iterate\_p_{jk_1} \in \bar{S}$	//The next iteration of the process instance p_{jk_1} is invoked.

Case (e) The invoked process is not an unfolded or in-zoomed subprocess of any process or object and is not an ancestor of the process $proc_i$. In this case, an instance of the process is created in a waiting state. In Figure 4.14, **proc[i]** invokes an independent process **proc[e]** such that a new instance of **proc[e]** is created as waiting.

(3.7.0) $\forall l.\ l \in S \wedge type(l) =$ "*invocation*" $\wedge\, l = (proc_i, proc_j)$	//The rule is defined for any invocation link l that belongs to the OPM graph.
(3.7.1) $\wedge(\neg\exists l_1.\ l_1 \in E \wedge (type(l_1) =$ "*in-zoomed*" $\vee\ type(l_1) =$ "$unfolded$") $\wedge\ l_1.destination =$ $proc_j)$	//The process $proc_j$ is independent (not a part or an operation of any other thing).
(3.7.2) $\wedge\, proc_j \notin ancestors(proc_i)$	//$proc_j$ is not an ancestor of $proc_i$.

(3.7.3) $\wedge\ \exists k_1 \in [1..K] \wedge \neg p_{jk_1}.isActive \wedge \neg p_{jk_1}.isWaiting$	//There exists an unused available process instance of $proc_j$.
(3.7.4) $\rightarrow ev\_invoke\_p_{jk_1}$.	//Then, a process invocation event is added to the result set. An event of $proc_j$ process instance creation is set to true.

4.5 StepI-StepII Transitions

The transition precondition for the τ_{step} transition relation is that the events set Ev is not empty and the configuration does not include an event of type *ev_terminate*. The transition is separated into two phases. The first phase of the transition deals with objects consumption, object instances creation, or their states modification. Then, through the second phase, process activation events are treated.

4.5.1 StepI Transition

4.5.1.1 *SysState* Transition

(4.0.0) $Ev \neq \oslash \wedge SysState =$ "$stepI$" $\wedge\ ev\_terminate \notin Ev$.
(4.0.1) $ev\_terminate \notin Ev \rightarrow SysState' =$ "$stepII$" .
(4.0.2) $ev\_terminate \in Ev \rightarrow SysState' =$ "end" .

4.5.1.2 Objects Consumption

Upon a process p_{ik} termination event, all the local objects that belong to the process scope should be cleaned away from the following configuration. The value of the object instances becomes ϵ, all the existing object attributes and components are consumed, and all the links connecting other objects/processes with the consumed objects are assigned to the $false$ value.

To summarize, the consumption mechanism works in the following way: (1) the process that consumes the object instance becomes active; (2) a

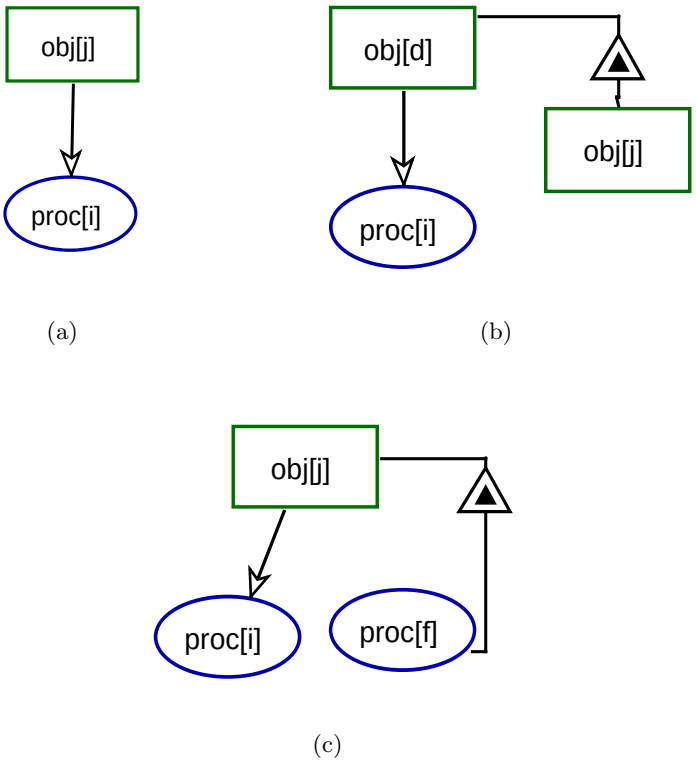


Figure 4.15: Object consumption rules exemplified.

subset of the process entry links becomes *true*, this includes the consumption links; (3) by the process termination, if the connected object still exists, that is no one of the process descendants has consumed it, the object is consumed with all the link instances connected to it and all its features. In Figure 4.15, consumption of an object **obj[j]** following the rules defined in (4.1.0) through (4.1.10) are exemplified. Case (a) describes consumption of the object upon process **proc[i]** termination following the consumption link connecting the object with the process. In case (b), the object is consumed because it is a feature of another object that is consumed. Case (c) illustrates consumption of an object operation that occurs while consuming of the object **obj[j]**.

(4.1.0) $ev\_term\_p_{ik} \in Ev \wedge$	//The rule is defined for any process termination event that belongs to the set Ev.
(4.1.2) $\forall j, k_1.\ j \in \|1..\|Objs\|\| \wedge k_1 \in \|1..K\| \wedge consumes_{ik,jk_1}$	//There exists an object instance that should be consumed by the terminating process;
$\vee(\exists d, k_2. \wedge (\exists l.\ l \in E \wedge ((type(l) = characterization \wedge characterization_{dk_2,jk_1}))))$	//Or the object is a feature of an object that is consumed by the process.
$\rightarrow consume\_o_{jk_1}\_p_{ik} \in Ev'$	//Then a consumption event of the object is added to the set Ev'.
(4.1.3) $\forall f, k_3.\ f \in \|1..\|Procs\|\| \wedge k_3 \in \|1..K\| \wedge consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge characterizes_{jk_1,fk_3} \wedge (p_{fk_3}.isActive \vee p_{fk_3}.isWaiting) \rightarrow halt\_p_{fk_3} \in Ev'$	//This rule terminates all the processes that are aggregated by the consumed objects.
(4.1.4) $\forall d, k_2.\ d \in \|1..\|Procs\|/\|Objs\|\| \wedge k_2 \in \|1..K\| \wedge \exists l.\ l \in E \wedge (consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) =$ "$result$" $\wedge\ result_{dk_2,jk_1} \rightarrow \neg result'_{dk_2,jk_1})$	//Termination of all the result link instances that enter the consumed object.

(4.1.5) $\forall d, k_2.\ d \in |1..|Procs|/|Objs|| \wedge k_2 \in |1..K| \wedge \exists l.\ l \in E(consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) = \text{“}characterization\text{”} \wedge characterize_{dk_2,jk_1} \rightarrow \neg characterize'_{dk_2,jk_1})$

//Termination of all the result links connected to the object.

(4.1.6) $\forall d, k_2.\ d \in |1..|Procs|/|Objs|| \wedge k_2 \in |1..K| \wedge \exists l.\ l \in E(consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) = \text{“}aggregation\text{”} \wedge aggregate_{dk_2,jk_1} \rightarrow \neg aggregate'_{dk_2,jk_1})$

//Termination of all the characterization links connected to the object through which the terminated object is characterized.

(4.1.7) $\forall d, k_2.\ d \in |1..|Procs|/|Objs|| \wedge k_2 \in |1..K| \wedge \exists l.\ l \in E(consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) = \text{“}instrument\text{”} \wedge instrument_{dk_2,jk_1} \rightarrow \neg instrument'_{dk_2,jk_1})$

//Termination of all the instrument links connected to the object.

(4.1.8) $\forall d, k_2.\ d \in |1..|Procs|/|Objs|| \wedge k_2 \in |1..K| \wedge \exists l.\ l \in E(consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) = \text{“}effect\text{”} \wedge effect_{dk_2,jk_1} \rightarrow \neg effect'_{dk_2,jk_1})$

//Termination of all the effect links connected to the object.

(4.1.9) $\forall d, k_2.\ d \in |1..|Procs|/|Objs|| \wedge k_2 \in |1..K| \wedge \exists l.\ l \in E(consume\_o_{jk_1}\_p_{ik} \in Ev' \wedge l = (thing_d, obj_j) \wedge type(l) = \text{“}consumption\text{”} \wedge consume_{dk_2,jk_1} \rightarrow \neg consume'_{dk_2,jk_1})$ //Termination of all the consumption links connected to the object.

(4.1.10) $consume\_o_{jk_1}\_p_{ik} \in Ev' \rightarrow o_{jk_1}.val[0]' = \epsilon \wedge o_{jk_1}.val[1]' = \epsilon$ //The terminated object instance becomes undefined in the next configuration.

4.5.1.3 Generating link instances that connect a newly created object with other entities

The following rules define transition of link variables connecting the new object instance o_{ik} with appropriate objects/processes via characterization or aggregation links. The instances are connected to the object o_{ik} within the *context* of the process that has created the object. In Figure 4.16, a subprocess **proc[j]** generates an object **obj[i]**, and by the process termination, the object is added to its parent, a process **proc[h]** via a characterization relation.

The following rule treats the case in which one of the ancestors of the process p_{jk_1} has a characterization relation with the object obj_i. The closest ancestor instance in the refinement tree, whose process has a characterization relation to the newly created object will hold characterization relation with the object instance.

Formally:

(4.2.0) $ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev \wedge$ //The rule is defined for any event of object creation that was generated by a terminating process.

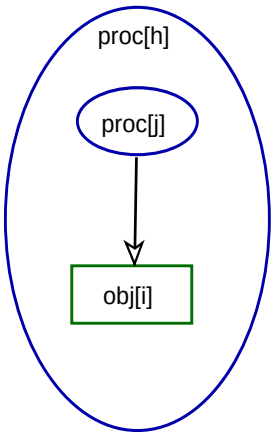


Figure 4.16: Adding a characterization relation among a newly created object and an ancestor process.

(4.2.1) $\exists h, k_2.\ h \in [1..\|Procs\|] \wedge k_2 \in \|1..K\| \wedge p_{hk_2} \in p_{jk_1}.ancestors \wedge \exists l.\ l = (proc_h, obj_i) \wedge type(l) =$ "$characterizes$"	//There exists an ancestor process p_h of the process p_j that has a characterization relation with the object o_i. In addition, the instance process p_{hk_2} is the ancestor instance of the process instance p_{jk_1};
(4.2.2) $\wedge\ (\forall d, k_3.\ d \in \|1..\|Procs\|\| \wedge k_3 \in \|1..K\| \wedge p_{dk_3} \in p_{jk_1}.ancestors \wedge p_{hk_2} \in p_{dk_3}.ancestors \wedge \neg(\exists l_1\ .l_1 \in E \wedge l_1 = (proc_d, obj_i) \wedge type(l_1) =$ "$characterizes$"$))$	//and the process ancestor p_h that has a characterization relation with the object o_i is the closest ancestor having such relation in the hierarchy of the process p_j ancestors.

(4.2.3) $\rightarrow characterizes'_{hk_2,ik}$. //If the conjunction of all the listed conditions (4.2.0) to (4.2.2) is satisfied, a characterization relation instance among the process p_{hk_2} and the object $o_{ik[s]}$ is generated.

In case the object is not an attribute or a component of any object or process, result links are added to all the ancestors of the creating process. In Figure 4.17, the newly created object **obj[i]** is added to the parent process of **obj[j]**.

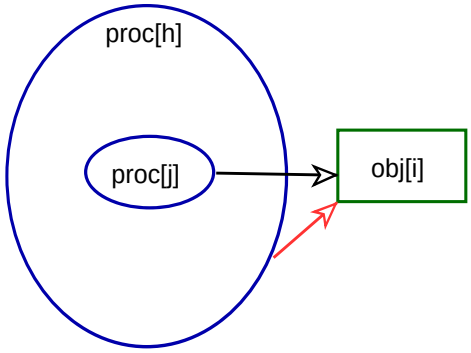


Figure 4.17: Adding an object instance to context of a parent process.

(4.3.0) $ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev$ //The rule is defined for any event specifying the creation of an object instance by a process.

(4.3.1) $\wedge \neg(\exists l.\ l \in StructLinks \wedge l.dest = obj_i \wedge type(l) \in \{\text{“}characterization\text{”}, \text{“}aggregation\text{”}\})$ //The object is not an attribute or a part of any object or process.

(4.3.2) $\rightarrow (\forall h, k_2.\ h \in \lvert 1..\lvert Procs\rvert\rvert \wedge k_2 \in \lvert 1..K\rvert.\ p_{hk_2} \in p_{jk_1}.ancestors \rightarrow result'_{hk_2,ik[s]})$.	//Add result link instances between all the ancestors of the process that creates a new object instance and the object instance.

In a more complex case, the object characterizes or is part of another object. The whole object or the exhibitor (the characterized object) already has an existing instance in the context of the process ancestors. In this case, an appropriate link among the object instances should be added, as the example in Figure 4.18 shows.

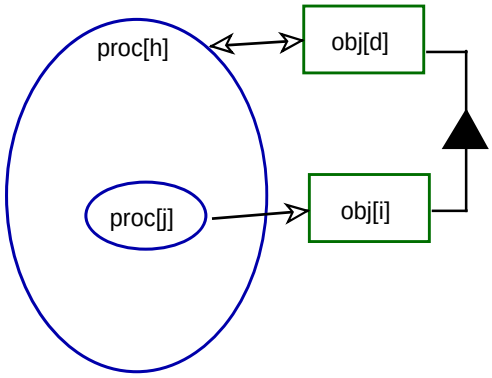


Figure 4.18: Adding an aggregation relation.

(4.4.0) $ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev \wedge$	//The rule is defined for any event specifying the creation of an object instance by a process.
(4.4.1) $\exists l.\ l \in StructLinks \wedge l = (obj_d, obj_i) \wedge type(l) \in \{\text{“}characterization\text{”}, \text{“}aggregation\text{”}\}$	//If there exists a characterization or aggregation relation from obj_d to obj_i.

(4.4.2) $\wedge\ \exists a, k_2, k_3.\ a \in |1..|Procs|| \wedge k_2 \in |1..K| \wedge k_3 \in |1..K| \wedge p_{ak_3} \in p_{jk_1}.ancestors \wedge$ //If process p_{jk_1} has ancestor p_{ak_3}.

(4.4.3) $(result_{ak_3,dk_2} \vee characterizes_{ak_3,dk_2} \vee effect_{ak_3,dk_2} \vee instrument_{ak_3,dk_2})$ //The ancestor p_{ak_3} has a result, characterization or effect link with the instance o_{dk_2} of the object obj_d.

(4.4.4) $\rightarrow (type(l) = \text{“}characterization\text{”} \wedge characterization'_{dk_2,ik}) \vee (type(l) = \text{“}aggregation\text{”} \wedge aggregation'_{dk_2,ik})$ //The appropriate relation instance among the objects is created.

(4.4.5) $(\forall b, k_4.\ b \in |1..Procs|| \wedge k_4 \in |1..K| \wedge p_{bk_4} \in p_{jk_1}.ancestors \wedge p_{a,k_3} \in p_{bk_4}.ancestors \wedge result'_{bk_4,ik}$ //A new object is added to the context of all the ancestors up to the process p_{a,k_3}.

If an object obj_i that holds a new object obj_j as its attribute does not exist, obj_j must be kept in the context of ancestors of the creating process until assembling time of obj_i. This is achieved by adding result link instances among the object, obj_j and all the ancestors up to the assembling process of the object obj_i. In Figure 4.19 the red link between **proc[h]** and **obj[i]** is added following the described rule.

(4.5.0) $ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev \wedge$ //The rule is defined for any event specifying creation of an object instance by a process.

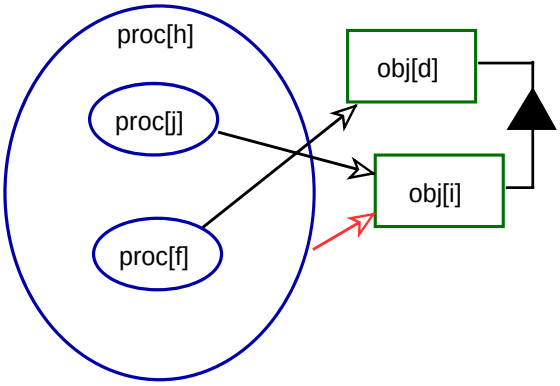


Figure 4.19: Holding an object in the parent process context using result link.

(4.5.1) $\exists l.\ l \in StructLinks \wedge l = (obj_d, obj_i) \wedge type(l) \in \{\text{“}characterization\text{”}, \text{“}aggregation\text{”}\} \wedge$ //There exists a characterization or an aggregation relation from obj_d to obj_i.

(4.5.2) $\exists a, k_2, k_3.\ a \in |1..|Procs|| \wedge k_2 \in |1..K| \wedge k_3 \in |1..K| \wedge p_{ak_3} \in p_{jk_1}.ancestors \wedge$ //The process p_{jk_1} has an ancestor p_{ak_3}.

The following rule defines assembling of a newly created object with the object component/attribute instances that already exist in the process environment.

(4.6.0) $ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev \wedge (effect_{jk_1,dk_2} \vee instrument_{jk_1,dk_2})$ //The rule is defined for any event specifying creation of an object instance by a process.

4.5.1.4 Invocation links derived by objects modification/creation events and timeout events

In the following cases we treat the invocation of processes following timeout and create/modify events that emanate from a process instance. The set includes the following four events: $ev\_max\_timeout\_p_{ik}$, $ev\_min\_timeout\_p_{ik}$, $ev\_create\_o_{dk_1[s]}\_p_{ik}$, and $ev\_modify\_o_{dk_1[s]}\_p_{ik}$.

Case (a) An event or a timeout link has a target that is one of the subprocesses enclosed by the process $proc_i$ in-zoomed ancestor. In addition, the target process itself is not one of the process $proc_i$ ancestors. Then, the event changing the current timeline of the closest common ancestor between $proc_i$ and the target process will be generated. This case is illustrated in Figure 4.20.

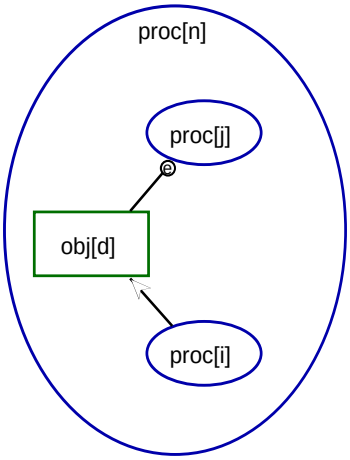


Figure 4.20: A process invocation following an internal event - example 1

Formally:

(4.7.0) $\exists l.\ l \in ProcLinks \wedge l.dest = proc_j \wedge isEvent(l) \wedge$ //There exists an event link entering $proc_j$.

(4.7.1) $(ev\_min\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) =$ "$minTimeout$")	//There is a min or max timeout event of a process $proc_i$.
(4.7.2) $\vee (ev\_min\_timeout\_o_{dk_1 s} \in Ev \wedge l.src = s \wedge type(l) =$ "$minTimeout$"	//There exists a timeout exception event from the object obj_d.
$\wedge \exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{ik,dk_1}) \wedge p_{ik}.isActive)$	// The object instance o_{dk_1} that is in state s and that caused the timeout exception is in the context of the process p_{ik}.
(4.7.3) $\vee (ev\_max\_timeout\_o_{dk_1 s} \in Ev \wedge l.src = s \wedge type(l) =$ "$maxTimeout$"$\wedge$	
$\exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{ik,dk_1}) \wedge p_{ik}.isActive)$	//A similar rule is defined for max timeout links.
(4.7.4) $\vee ((ev\_create\_o_{dk_1[s]}\_p_{ik} \in Ev \vee ev\_modify\_o_{dk_1[s]}\_p_{ik} \in Ev) \wedge (l.src = s \vee l.src = obj_d)$	//There exists an event link from an object or its state to the process and an appropriate event was fired.
$\wedge \exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{ik,dk_1}) \wedge p_{ik}.isActive)$	// The object instance o_{dk_1} that caused the timeout exception is in the context of the process p_{ik}.
(4.7.5) $\wedge proc_j \notin ancestors(proc_i)$	//$proc_j$ is not an ancestor of $proc_i$. The two processes $proc_i$ and $proc_j$ have no in-zoomed or unfolded path between them.

(4.7.6) $\wedge \exists n.\ n \in [1..\|Procs\|] \wedge proc_n \in ancestors(proc_i) \wedge proc_n \in ancestors(proc_j)$	//The destination and target processes have a common ancestor $proc_n$
(4.7.7) $\wedge \forall h.\ h \in [1..\|Procs\|] \wedge h \neq n \wedge proc_h \in ancestors(proc_i) \wedge proc_h \in ancestors(proc_j) \wedge proc_h \in ancestors(proc_n)$	//The ancestor $proc_n$ is the nearest common ancestor of the two processes in the refinement tree.
(4.7.8) $\wedge \exists l_2.\ l_2 \in ProcLinks \wedge l_2 = (proc_n, proc_j) \wedge type(l_2) =$ "*in-zooming*"	//The target process is enclosed by the process $proc_n$ in-zoomed diagram.
(4.7.9) $\wedge \exists k_2.\ k_2 \in [1..K] \wedge p_{nk_2} \in p_{ik}.ancestors$	//Find an ancestor instance in whose context the terminating process p_{ik} runs.
(4.7.10) $\rightarrow ev\_change\_t_{partOrder(proc_j)}\_p_{nk_2}$	//The timeline of the process p_{nk_2} is changed from the current timeline to the timeline to which the process $proc_j$ belongs. Keep the object in the process context.
$\_t_{p_{nk_2}.excSeqNum} \in Ev' \wedge ((ev\_create\_o_{dk_1[s]}\_p_{ik} \vee ev\_modify\_o_{dk_1[s]}\_p_{ik}) \in Ev \rightarrow event'_{dk_1,nk_2})$	

Case (b) A timeout or event link has a target process that is a subprocess of an active process or an attribute of an existing object. In addition, the invoked process is not enclosed by any in-zoomed process. This case is illustrated in Figure 4.21.

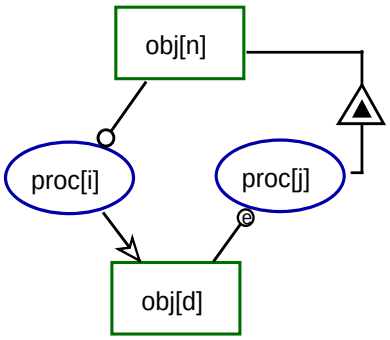


Figure 4.21: A process invocation following an internal event - example 2

(4.8.0) $\exists l.\ l \in ProcLinks \wedge l.dest = proc_j \wedge isEvent(l) \wedge$	//The rule is invoked for any event link entering a process $proc_j$
(4.8.1) $((ev\_max\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) =$ "$maxTimeout$")	//If the link l is a maximum timeout exception link with a process $proc_i$ source.
(4.8.2) $\vee\ (ev\_min\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) =$ "$minTimeout$")	//If the link l is a minimum timeout exception link with a process source.

(4.8.3) $\vee (\exists i, k.\ i \in \|1..\|Procs\| \wedge k \in \|1..K\| \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive)$	//There exists an object obj_d in the context of $proc_i$.
$\wedge((ev\_max\_timeout\_o_{dk_1 s} \in Ev \wedge (l.src = s) \wedge type(l) =$ "$maxTimeout$") $\vee$ $(ev\_min\_timeout\_o_{dk_1 s} \in Ev \wedge (l.src = s) \wedge type(l) =$ "$minTimeout$")	//The current state s of the object has reached its mininimum or maximum time boundary.
$\vee(ev\_create\_o_{dk_1[s]\_p_{ik}} \in Ev \wedge (l.src = s \vee l.src = obj_d))$	//There is an event link from state s or from the object obj_d and an event of object creation (or creation in the sate s) has occurred.
$\vee(ev\_modify\_o_{dk_1 s\_p_{ik}} \in Ev \wedge (l.src = s \vee l.src = obj_d)))$	//An event of an object entering state s has occurred.
(4.8.4) $\wedge proc_j \notin ancestors(proc_i)$	//In addition, $proc_j$ is not an ancestor of $proc_i$.
(4.8.5) $\wedge \exists n.\ n \in [1..\|Procs\|] \wedge thing_n \in ancestors(proc_i) \wedge thing_n \in ancestors(proc_j)$	//The destination and the target processes have a common ancestor $thing_n$.
(4.8.6) $\wedge \forall h.\ h \in [1..\|Procs\|/\|Objs\|] \wedge h \neq n \wedge thing_h \in ancestors(proc_i) \wedge thing_h \in ancestors(proc_j) \wedge thing_h \in ancestors(thing_n)$	//The ancestor $thing_n$ is the closest common ancestor of the two processes in the refinement tree.

(4.8.7) $\wedge \neg\exists l_1.\ l_1 \in ProcLinks \wedge l_1 = (thing_n, proc_j) \wedge type(l_1) =$ "*in-zooming*"	//Then, if the target process is not enclosed by the process $proc_n$ in-zoomed diagram,
(4.8.8) $\wedge\ \exists k_2.\ k_2 \in [1..K] \wedge thing_{nk_2} \in p_{ik}.ancestors$	//find the ancestor instance in whose context process p_{ik} runs
(4.8.9) $\wedge\ \exists k_3 \in [1..K] \wedge \neg p_{jk_3}.isActive \wedge \neg p_{jk_3}.isWaiting$	//Find a non active process $proc_j$ instance.
(4.8.10) $\rightarrow$ $ev\_invoke\_p_{jk_3}\_thing_{nk_2} \in Ev'$	//An event of a process invocation is added to the set Ev'.
(4.8.11) $\wedge((ev\_max\_timeout\_p_{ik} \vee ev\_min\_timeout\_p_{ik}) \rightarrow ev\_halt\_pik \in Ev')$	//Additionally, halt the process if its timeout event is true and all the listed preconditions are satisfied.

Case (c) An invocation link from the process $proc_i$ enters a subprocess of a currently active process or characterizes an existing object. In addition, the invoked process is not enclosed by any in-zooming processes and has no common ancestor with the process $proc_i$. This case is illustrated in Figure 4.22.

(4.9.0) $\exists l, j.\ l \in ProcLinks \wedge j \in \|1..\|Procs\|\| \wedge l.dest = proc_j \wedge isEvent(l)$	//There exists an event link entering process $proc_j$.

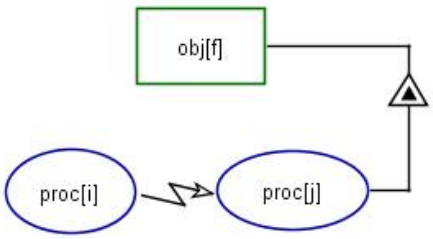


Figure 4.22: A process invocation following an internal event - example 3

(4.9.1) $\wedge ((ev\_max\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) = \text{“}maxTimeout\text{”}) \vee (ev\_min\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) = \text{“}minTimeout\text{”})$

//The max or min timeout exception event from p_{ik} is in the set Ev.

(4.9.2) $\vee (ev\_min\_timeout\_o_{dk_1 s} \in Ev \wedge l.src = s \wedge type(l) = \text{“}minTimeout\text{”} \wedge \exists i, k. \quad i \in |1..|Procs|| \wedge k \in |1..K| \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive))$

//The min timeout exception event from object o_{dk} or from its state s is in the set Ev and there exists a procedural link instance from the process p_{ik} to the object whose timeout event is treated.

(4.9.3) $\vee (ev\_max\_timeout\_o_{dk_1 s} \in Ev \wedge l.src = s \wedge type(l) = \text{“}maxTimeout\text{”} \wedge \exists i, k. \quad i \in |1..|Procs|| \wedge k \in |1..K| \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive)$

//The max timeout exception event from object o_{dk} or from its state s is in the set Ev and there exists a procedural link instance from the process p_{ik} to the object whose timeout event is treated.

(4.9.4) $\vee (ev\_create\_o_{dk_1[s]}\_p_{ik} \in Ev \wedge (l.src = s \vee l.src = obj_d))$ $\vee (ev\_modify\_o_{dk_1[s]}\_p_{ik} \in Ev \wedge (l.src = s \vee l.src = obj_d))$	//There exists an event of object o_{dk_1} creation (at state s). //There exists an event of object o_{dk_1} (entering the state s).
(4.9.5) $\wedge proc_j \notin ancestors(proc_i)$	//$proc_j$ is not an ancestor of $proc_i$.
(4.9.6) $\wedge \neg\exists n.\ n \in [1..\|Procs\|] \wedge thing_n \in ancestors(proc_i) \wedge thing_n \in ancestors(proc_j)$	//The destination and target processes have no common ancestor.
(4.9.7) $\wedge\ \forall l_1.\ l_1 \in E \wedge l_1.destination = proc_j \wedge type(l_1) \neq$ "*in-zooming*"	//The invoked process $proc_j$ is not enclosed by any in-zooming diagram.
(4.9.8) $\wedge\ \exists l_2.\ l_2 \in E \wedge type(l_2) =$ "$unfolding$" $\wedge\ l_2 = (thing_f, proc_j) \wedge \exists k_1 \in \|1..K\| \wedge (p_{fk_1}.isActive \vee o_{fk_1}.val \neq (\epsilon, \epsilon))$	//There exists an instance of $thing_f$ that is active and whose object or process class has an unfolding relation with $proc_j$.
(4.9.9) $\wedge\ \exists k_2 \in [1..K] \wedge \neg p_{jk_2}.isActive \wedge \neg p_{jk_2}.isWaiting$	//The process $proc_j$ has unused instance identifier.
(4.9.10) $\rightarrow ev\_invoke\_p_{jk_2}\_thing_{fk_1} \in Ev'$	//Then, an event adding the process instance invocation by an OPM thing is added to the set Ev'.

(4.9.11) $\wedge((ev\_max\_timeout\_p_{ik} \vee ev\_min\_timeout\_p_{ik}) \rightarrow ev\_halt\_pik \in Ev')$ $\wedge((ev\_create\_o_{dk_1[s]}\_p_{ik} \vee ev\_modify\_o_{dk_1[s]}\_p_{ik}) \in Ev \rightarrow event'_{dk_1,jk_2})$.	//Halt the process from which the timeout event emanates.

Case (d) If the event link holds as a target one of the process $proc_i$ ancestors, an *ev_iterate* event will be generated. This case is illustrated in Figure 4.23.

(4.10.0) $\exists l, j.\ l \in ProcLinks \wedge j \in \lvert 1..\lvert Procs\rvert\rvert \wedge l.dest = proc_j \wedge isEvent(l)$	//There exists an event link that invokes $proc_j$.
(4.10.1) $\wedge((ev\_max\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) = \text{“}maxTimeout\text{”}) \vee (ev\_min\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) = \text{“}minTimeout\text{”})$	//l is a timeout link and the related event was fired.
(4.10.2) $\vee(\exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive$	//There exists a link that connects an object obj_d with a process $proc_i$.
$\wedge((ev\_min\_timeout\_o_{dk_1s} \in Ev \wedge l.src = s \wedge type(l) = \text{“}minTimeout\text{”})$	//The link is a minimum timeout event link that is connected to obj_d and the related timeout event was fired.

$\vee(ev\_max\_timeout\_o_{dk_1s} \in Ev \wedge l.src = s \wedge type(l) =$ "$maxTimeout$")	//The link is a maximum timeout event link that is connected to obj_d and the related maximum timeout event was fired.
$\vee(ev\_create\_o_{dk_1s}\_p_{ik} \in Ev \wedge l.src = s)$	//An event of obj_d creation in state s is in the set Ev.
$\vee(ev\_modify\_o_{dk_1s}\_p_{ik} \in Ev \wedge l.src = s)$	//An event of obj_d entering state s is in the set Ev.
$\vee(ev\_create\_o_{dk_1}\_p_{ik} \in Ev \wedge l.src = obj_d)$	//An event of obj_d creation is in the set Ev.
$\vee(ev\_modify\_o_{dk_1}\_p_{ik} \in Ev \wedge l.src = obj_d))$	//An event of obj_d modification is in the set Ev.

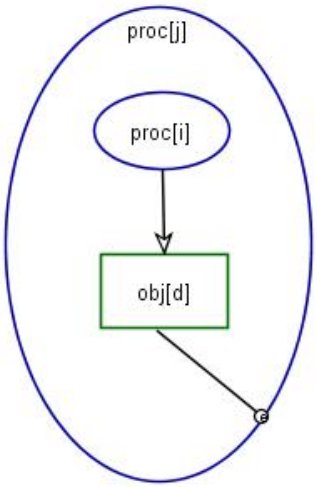


Figure 4.23: A process invocation following an internal event - 4

(4.10.3) $\wedge p_{jk_2} \in p_{ik}.ancestors$	//$proc_j$ is an ancestor of $proc_i$.

(4.10.4) $\rightarrow ev\_iterate\_p_{jk_2} \in Ev'$	//Then an event of $proc_j$ iterating is added the set Ev'.
$\wedge((ev\_create\_o_{dk_1[s]}\_p_{ik} \vee ev\_modify\_o_{dk_1[s]}\_p_{ik}) \in Ev \rightarrow event'_{dk_1,jk_2})$.	//The value of an event link between an object obj_d and the iterating process $proc_j$ becomes true in the next configuration if in the current configuration an event of the object modifying or creation is processed.

Case (e) The invoked process is not an unfolding or an in-zooming subprocess of any process or object and is not an ancestor of the process $proc_i$. This case is illustrated in Figure 4.24.

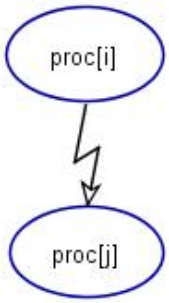
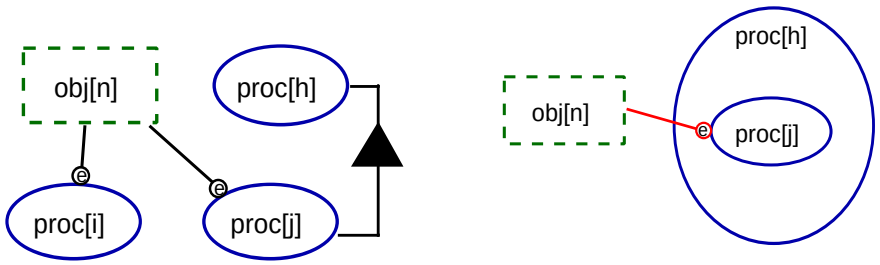


Figure 4.24: A process invocation following an internal event - example 5

(4.11.0) $\exists l, j.\ l \in ProcLinks \wedge j \in \lvert 1..\lvert Procs\rvert\rvert \wedge l.dest = proc_j \wedge isEvent(l)$	//There exists an event link entering $proc_j$.
(4.11.1) $\wedge((ev\_max\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) =$ "$maxTimeout$")	//A maximum timeout event corresponding to the link and holding $proc_i$ as its source is present in the set Ev.
$\vee(ev\_min\_timeout\_p_{ik} \in Ev \wedge l.src = proc_i \wedge type(l) =$ "$minTimeout$")	//A minimum timeout event corresponding to the link and holding $proc_i$ as its source is present in the set Ev.
$\vee(ev\_min\_timeout\_o_{dk_1s} \in Ev \wedge l.src = s \wedge type(l) =$ "$minTimeout$" $\wedge\ \exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive))$	//A minimum timeout event corresponding to the link belongs to an object obj_d and is in Ev.
$\vee(ev\_max\_timeout\_o_{dk_1s} \in Ev \wedge l.src = s \wedge type(l) =$ "$maxTimeout$" $\wedge\ \exists i, k.\ i \in \lvert 1..\lvert Procs\rvert\rvert \wedge k \in \lvert 1..K\rvert \wedge (result_{ik,dk_1} \vee characterizes_{jk,dk_1} \vee effect_{jk,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive)$	//A maximum timeout event corresponding to the link belongs to an object obj_d.
$\vee(ev\_create\_o_{dk_1s}\_p_{ik} \in Ev \wedge l.src = s \wedge isEvent(l))$	//An event of object creation in state s is in Ev and it corresponds to the link l.
$\vee(ev\_modify\_o_{dk_1s}\_p_{ik} \in Ev \wedge l.src = s \wedge isEvent(l))$	//An event of entering state s by an object is in the set Ev and it corresponds to the link l.

$\vee(ev\_create\_o_{dk_1[s]}\_p_{ik} \in Ev \wedge l.src = obj_d \wedge isEvent(l))$	//An object creation event in Ev corresponds to the link l.
$\vee(ev\_modify\_o_{dk_1[s]}\_p_{ik} \in Ev \wedge l.src = obj_d \wedge isEvent(l)))$	//An event of object modification is in the set Ev and it corresponds to the link l.
(4.11.2) $\wedge$ $(\neg\exists l_1.\ l_1 \in E \wedge (type(l_1) =$ "*in-zooming*" $\vee$ $type(l_1) =$ "$unfolding$") $\wedge$ $l_1.destination = proc_j)$	//The $proc_j$ process triggered by the event link is not a subprocess or a feature of any OPM thing.
(4.11.3) $\wedge$ $proc_j \notin ancestors(proc_i)$	//The processes $proc_i$ and $proc_j$ are not ancestors of each other.
(4.11.4) $\wedge$ $\exists k_2 \in [1..K] \wedge \neg p_{jk_2}.isActive \wedge \neg p_{jk_2}.isWaiting$	//There exists an unused instance of process $proc_j$.
(4.11.5) $\rightarrow ev\_invoke\_p_{jk_2} \in Ev'$	//Then, a new instance of $proc_j$ will be created.
$\wedge((ev\_create\_o_{dk_1[s]}\_p_{ik} \vee ev\_modify\_o_{dk_1[s]}\_p_{ik}) \in Ev \rightarrow event'_{dk_1,jk_2})$.	//A value of a variable representing the event link between the triggering event object and the created process will become true in the next configuration.

Case (f) In the following case we treat invocation of processes following timeout events that emanate from an object instance, where the object is not in the context of any existing process. The possible events are: $ev\_max\_timeout\_o_{ik}$ and $ev\_min\_timeout\_o_{ik}$. This situation is legal only if an invoked process is not a subprocess of an in-zoomed process. In Figure 4.25 (a) includes two legal options, both **proc[i]** and **proc[j]** can be invoked by **obj[n]**; (b) describes illegal option in which the process **proc[j]** is a subprocess



(a) (b)

Figure 4.25: Invocation of a process by an environmental object.

of an in-zoomed process and cannot be triggered by an object that is not in the **proc[h]** context.

(4.12.0) $\exists l, d, j.\ l \in ProcLinks \wedge d \in \|1..\|Objs\|\| \wedge j \in \|1..\|Procs\|\| \wedge l.dest = proc_j \wedge isEvent(l)$	//The rule is described for any event link l that belongs to G.
(4.12.1) $\wedge$ $((ev\_min\_timeout\_o_{dk_1}s \in Ev \wedge l.src = s \wedge type(l) =$ "$minTimeout$" $\wedge \neg(\exists i, k.\ i \in \|1..\|Procs\|\| \wedge k \in \|1..K\| \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive))\vee$	//The link has an object state s as its source and the link type is min-timeout event link and also timeout event related to s is set on and the object is not in the context of any active process $proc_i$. The last condition means that the object is not in the context of any active process.
$(ev\_max\_timeout\_o_{dk_1}s \in Ev \wedge l.src = s \wedge type(l) =$ "$maxTimeout$"$\wedge$	//The link holds an object state s as its source and the link type is max timeout event.

$\neg(\exists i,k.\ i \in \|1..\|Procs\| \wedge k \in \|1..K\| \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{jk,dk_1}) \wedge p_{ik}.isActive)) \vee$	//Timeout event related to s is on and obj_d is not in the context of any active process $proc_i$.
$\vee(ev\_create\_o_{dk_1}[s] \in Ev \wedge (l.src = obj_d \vee l.src = s) \wedge (type(l) =$ "*instrument*" $\vee\ type(l) =$ *consumption*$) \wedge isEvent(l) \wedge \neg(\exists i,k.\ i \in \|1..\|Procs\| \wedge k \in \|1..K\| \wedge (result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{ik,dk_1}) \wedge p_{ik}.isActive)))$	//The link holds an object state s or the object obj_d as its source and it is an instrument-event or a consumption-event link and the object obj_d is not in the context of any active process.
(4.12.2) $\exists k_2.\ k_2 \in \|1..K\| \wedge \neg p_{jk_2}.isActive \wedge \neg p_{jk_2}.isWaiting$	//There exists an unused instance of $proc_j$.
(4.12.3) $\wedge\neg\exists l_1.\ l_1 \in E \wedge type(l_1) =$ "*in-zoomed*" $\wedge\, l_1.dest = proc_j$	//$proc_j$ is not a subprocess of any in-zoomed process.
(4.12.4) $\wedge((\exists l_2.\ l_2 \in E \wedge type(l_2) =$ "*unfolded*" $\wedge\, l_2 = (proc_i, proc_j)$	//There exists an active process $proc_i$ and the process is unfolded and $proc_j$ is its subprocess.
$\wedge\exists k_3.\ k_3 \in \|1..K\| \wedge p_{ik_3}.isActive \rightarrow ev\_invoke\_p_{jk_1}\_p_{ik_2} \in Ev' \wedge event'_{dk_1,jk_2})$	//In addition, the process $proc_i$ is active, then an event of invocation of $proc_j$ in the context of $proc_i$ will be inserted into Ev'.
(4.12.5) $\vee\ (\neg\exists l_2.\ l_2 \in E \wedge (type(l_2) =$ "*unfolded*" $\vee\ type(l_2) =$ "*in-zoomed*"$) \wedge l_2 = (proc_i, proc_j) \rightarrow$	//$proc_j$ is not a subprocess of any process.

$ev\_invoke\_p_{jk_1} \in Ev' \wedge event'_{dk_1,jk_2})$.	//Process $proc_j$ is invoked and an event link that connects obj_d with the process is created.

Case (g) In the following rules we treat illegal invocation of in-zoomed processes following events that source an object instance, where the object is not in the context of a process ancestor. This case was illustrated in Figure 4.25(b).

(4.13.0) $\exists l, i, j, d, k, k_1.\ i, j \in \|1..\|Procs\|\| \wedge d \in \|1..\|Objs\|\| \wedge k, k_1 \in \|1..K\| \wedge l \in ProcLinks \wedge (l.src = obj_d \vee l.src = s) \wedge l.dest = proc_j \wedge isEvent(l) \wedge ((result_{ik,dk_1} \vee characterizes_{ik,dk_1} \vee effect_{ik,dk_1} \vee instrument_{ik,dk_1}) \wedge p_{ik}.isActive \wedge proc_i \notin ancestors(proc_j))$	//There exists an event link l that has destination $proc_j$ and source obj_d or its state s. In addition, there exists an active process $proc_i$ that holds the object in its context and the process $proc_i$ is not one of $proc_j$ ancestors.
(4.13.1) $\wedge$ $((ev\_min\_timeout\_o_{dk_1[s]} \in Ev \wedge type(l) = \text{“}minTimeout\text{”}) \vee$ $(ev\_max\_timeout\_o_{dk_1[s]} \in Ev \wedge type(l) = \text{“}maxTimeout\text{”}) \vee$	// The min/max-timeout event related to the object obj_d has occurred and also l is of type “min/maxTimeout”,
$(ev\_create\_o_{dk_1[s]} \in Ev))$	//or an event of object creation is on.
(4.13.2) $\wedge \exists l_1.\ l_1 \in E \wedge type(l_1) =$ “*in-zoomed*” $\wedge\ l_1.dest = proc_j$	//$proc_j$ is enclosed by in-zoomed process.
(4.13.3) $\rightarrow ev\_terminate \in Ev'$.	//The event is illegal and the system should be terminated.

(4.14.0)$ev\_max\_timeout\_o_{dk_1[s]} \in Ev \rightarrow \neg(ev\_max\_timeout\_o'_{dk_1[s]})$	//The event is treated and set to false for the next configuration.
(4.15.0)$ev\_min\_timeout\_o_{dk_1[s]} \in Ev \rightarrow \neg(ev\_min\_timeout\_o'_{dk_1[s]})$	//The event is treated and set to false for the next configuration.

Case (h) In the following rule process current timeline is updated for all the active in-zoomed processes. The updating event is generated only if it does not contradict with other existing events.

(4.16.0) $ev\_term\_p_{ik} \in Ev \wedge p_{jk_1} \in p_{ik}.ancestors \wedge \exists l.\ l \in E \wedge type(l) =$ "*in-zoomed*" $\wedge\ l = (proc_j, proc_i)$	//There exists an in-zoomed process $proc_j$ whose subprocess is terminated.
(4.16.1) $\forall d, k_2.\ d \in [1..\lvert Procs\rvert] \wedge k_2 \in [1..K] p_{jk_1} \in p_{dk_2}.ancestors \wedge \exists l.\ l \in E \wedge type(l) =$ "*in-zoomed*" $\wedge\ l = (proc_j, proc_d) \wedge \neg p_{dk_2}.isActive \wedge \neg p_{dk_2}.isWaiting \wedge ((partialOrder(proc_i) + 1) \leq proc_j.TimelinesNumber)$	//The process $proc_j$ was the last active process in the current execution order of the process $proc_j$. In addition, there is no subprocess in a waiting state that belongs to $proc_j$. $proc_j$ did not reach its last execution timeline.

If all the listed conditions are satisfied, an event triggering a change in the execution timeline of the process $proc_j$ is added to a set Ev'. Formally:

$ev\_change\_t_{partialOrder(proc_i)+1}\_p_{jk_1}\_t_{p_{jk_1}} \cdot executionSeqNum \in Ev'$

4.5.2 StepII Transition

StepII transition handles all the deterministic steps of the system that are triggered by events generated in the previous steps *event* and *stepI*. The transition is partitioned into few subtransitions that cannot be executed simultaneously. The first subtransition is used to remove all the contradicting

events. Then, in the second subtransition, events of execution order modifying type and of process iterating type are treated. These events may trigger halting of processes and creation of new processes. In the following subtransition processes, halting is treated. Then, creation of new processes is treated. Processes are created in a waiting state. Finally, the last subtransition in *stepII* goes over all the waiting processes and determines those that can be invoked. These processes become active. The *stepII* transition is shown in Figure 4.26.

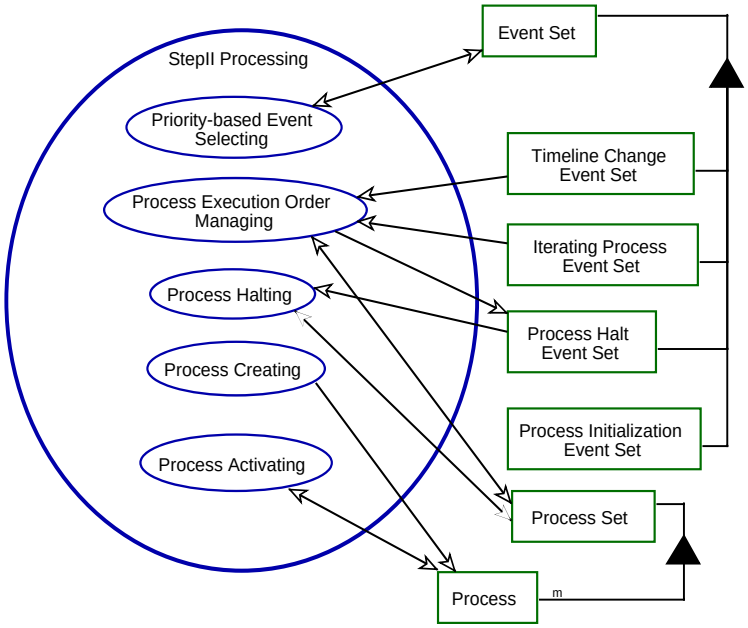


Figure 4.26: A process invocation following an internal event - example 6

4.5.2.1 *SysState* transition

(4.17.0) $Ev \neq \oslash \wedge SysState =$ "$stepII$" $\wedge\, ev\_terminate \notin Ev$.	//This is a precondition for invocation of all the transition rules related to "stepII".
(4.17.1) $SysState' =$ "$tick$"; .	//Next cycle of the system will be invoked after $stepII$.

4.5.2.2 Selection of events

(4.18.0) $(ev\_change\_t_a\_p_{ik}\_t_v \in Ev$	//If an event of change in the execution order of $proc_i$ is on,
$\vee ev\_iterate\_p_{ik} \in Ev)$	//or $proc_i$ iterating event is on.
$\wedge(ev\_change\_t_a\_p_{jk_1}\_t_v \in Ev$ $\vee ev\_iterate\_p_{jk_1} \in Ev)$	//If one of the two listed event types exists for $proc_j$ as well,
$\wedge p_{jk_1} \in p_{ik}.ancestors \rightarrow$	//and if $proc_j$ is ancestor of $proc_i$, then
$(ev\_change\_t_a\_p_{ik}\_t_v \in Ev \rightarrow ev\_change\_t_a\_p_{ik}\_t_v \notin Ev') \wedge (ev\_iterate\_p_{ik} \in Ev \rightarrow ev\_iterate\_p_{ik} \notin Ev'))$	//Execution order of the ancestor has higher priority, hence any events related to $proc_i$ are set off.
(4.19.0) $(ev\_change\_t_a\_p_{ik}\_t_v \in Ev \vee ev\_iterate\_p_{ik} \in Ev \vee ev\_invoke\_p_{ik} \in Ev$	//There is an execution order or iterating or invocation event of $proc_i$.

$) \wedge ev\_maxTimeout\_p_{jk_1} \in Ev' \wedge$	//There is also ma-timeout event related to $proc_j$
$p_{jk_1} \in p_{ik}.ancestors \rightarrow$	//$proc_j$ is ancestor of $proc_i$
$(ev\_change\_t_a\_p_{ik}\_t_v \notin Ev' \vee ev\_iterate\_p_{ik} \notin Ev' \vee ev\_invoke\_p_{ik} \notin Ev')$	//Max and min timeout events of a process ancestor have higher priority than any other execution order modifying events of the process.

4.5.2.3 Change in a process execution order

(4.20.0) $ev\_change\_t_1\_p_{ik}\_t_2 \in Ev \wedge$	//There exists event modifying current execution order of $proc_i$.
(4.20.1) $\forall j, k_1.\ j \in [1..\|Procs\|] \wedge p_{ik} \in p_{jk_1}.ancestors \wedge$	//For all the processes that hold $proc_i$ as their ancestor.
$\exists l.\ l \in E \wedge (type(l) = \text{“}in\text{-}zoomed\text{”}) \wedge l = (proc_i, proc_j) \rightarrow ev\_halt\_p_{jk_1}$	//If $proc_j$ is an in-zooming subprocess of $proc_i$, it must be halted.
(4.20.2) $p_{ik}.executionSeqNum' = t_1 \wedge$	//The execution order of $proc_i$ is updated.
$\forall l_1.\ l_1 \in ProcLinks \wedge type(l_1) = \text{“}in\text{-}zoomed\text{”} \wedge l_1 = (proc_i, proc_g) \wedge partialOrder(proc_g) = t_1 \wedge$	//For all the subprocesses of $proc_i$ that are located in the area of a new execution time line of $proc_i$,

(4.20.3) $\exists k_2.\ k_2 \in [1..K] \wedge \neg p_{gk_2}.isActive \wedge \neg p_{gk_2}.isWaiting \rightarrow ev\_create\_p_{gk_2} \in Ev'$.	//if an unused instance of the subprocess exists, an event of creating of the instance is added to Ev'.

4.5.2.4 Iteration of Processes

(4.21.0) $ev\_iterate\_p_{ik} \in Ev \wedge$	//An event of $proc_i$ iteration is in Ev.
(4.21.1) $p_{ik}.executionSeqNum' = 0 \wedge$	//Set the process execution order to zero.
$\forall l_1.\ l_1 \in ProcLinks \wedge type(l_1) =$ "*in-zoomed*" $\wedge l_1 = (proc_i, proc_g) \wedge partialOrder(proc_g) = 0 \wedge$	//For all the subprocesses of $proc_i$ that have partial order zero,
(4.21.2) $\exists k_2.\ k_2 \in [1..K] \wedge \neg p_{gk_2}.isActive \wedge \neg p_{gk_2}.isWaiting \rightarrow ev\_create\_p_{gk_2} \in Ev'$.	//if there exists an unused instance of the subprocess, add an instance creating event to the set Ev'.

4.5.2.5 Halting of Processes

(4.22.0) $ev\_halt\_p_{ik} \in Ev \rightarrow \forall j, k_2.\ j \in [1..\lvert Procs\rvert] \wedge k_2 \in [1..K] \wedge$	//There exists an event of halting of the process $proc_i$.

$(unfolding_{ik,jk_2} \vee in\text{-}zooming_{ik,jk_2}) \rightarrow ev\_halt\_p_{jk_2} \in Ev' \wedge$	//All its subprocesses are to be halted. The rules 4.22.0-4.23.0 are repeated while there exists any process halting event.
$\neg p_{ik}.isActive' \wedge \neg p_{ik}.isWaiting'$	//In the next configuration $proc_i$ is neither active nor waiting.
(4.22.1) $\forall d, k_2 \; .d \in \lvert 1..\lvert Objs \rvert \rvert \wedge k_2 \in [1..K] \wedge characterization_{ik,dk_2} \rightarrow (ev\_consume\_o_{dk_2}\_p_{ik} \in Ev' \; .$	//All the local objects that belong to the halting process are to be removed.
(4.22.3) $\wedge (\exists l. \; l \in E \wedge$	
$((type(l) = characterization \wedge characterization_{dk_2,jk_1})$	
$\rightarrow consume\_o_{jk_1}\_o_{dk_2} \in Ev' \wedge$	//This is a recursive rule, terminating all the object attributes.
(4.22.4)$\forall f, k_3. \; f \in \lvert 1..\lvert Procs \rvert \rvert \wedge k_3 \in \lvert 1..K \rvert \wedge characterizes_{jk_1,fk_3} \wedge$	//Next, the object operations must be halted.
$(p_{fk_3}.isActive \vee p_{fk_3}.isWaiting) \rightarrow halt\_p_{fk_3} \in Ev'$	//Terminate all the operations of the consumed objects.
(4.22.5) $consume\_o_{jk_1}\_thing_{ik} \in Ev \wedge$	//For any event of object consumption:

$\forall result_{dk_2,jk_1} \rightarrow \neg result'_{dk_2,jk_1})$	//all the result links are removed.
$\wedge\forall characterize_{x,y} \rightarrow \neg characterize'_{x,y})$	//All the characterization links are removed. x or y stand for jk_1.
$\wedge\forall aggregate_{x,y} \rightarrow \neg aggregate'_{x,y})$	//All the aggregation links are removed. x or y stand for jk_1.
$\wedge\forall instrument_{dk_2,jk_1} \rightarrow \neg instrument'_{dk_2,jk_1})$	//All the instrument links are removed.
$\wedge\forall effect_{dk_2,jk_1} \rightarrow \neg effect'_{dk_2,jk_1})$	//All the effect links are removed.
$\wedge\forall consume_{dk_2,jk_1} \rightarrow \neg consume'_{dk_2,jk_1}))$	//All the consumption links are removed.
$\wedge o_{jk_1}.val[0]' = \epsilon \wedge o_{jk_1}.val[1]' = \epsilon$	// The object becomes not existing.

4.5.2.6 Processes Creation

(4.23.0) $ev\_create\_p_{ik} \in Ev \wedge p_{ik}.isWaiting'$	//$proc_i$ is created in a waiting state.
(4.23.1) $\wedge\ thing_{jk_1} \in p_{ik}.ancestors \wedge (\exists l.\ l \in E \wedge l = (thing_j, proc_i) \wedge ((type(l) =$ "$in\text{-}zoomed$" $\rightarrow in\text{-}zoomed'_{jk_1,ik}) \vee (type(l) =$ "$unfolded$" $\rightarrow unfolded'_{jk_1,ik})))$	//Add a link to the in-zoomed or unfolded parent within whose context $proc_i$ is created.

4.5.2.7 Process Invocation

(4.24.0) $\forall i, k.\ i \in [1..\|Procs\|] \wedge k \in [1..K] \wedge p_{ik}.isWaiting \wedge$	//For all the waiting processes,
(4.24.1) $\exists j, k_1.\ j \in [1..\|Thing\|] \wedge k_1 \in [1..K] \wedge (in\text{-}zooming_{jk_1,ik} \vee unfolding_{jk_1,ik}) \wedge$	//If there exists a parent process $proc_j$,
(4.24.2) $\forall l \in E \wedge (l = (obj_d, proc_i) \vee (l = (s, proc_i) \wedge s \in states(obj_d))) \wedge \exists k_2.\ k_2 \in [1..K] \wedge (consume_{jk_1,o_{dk_2}} \vee effect_{jk_1,o_{dk_2}} \vee instrument_{jk_1,o_{dk_2}} \vee characterizes_{jk_1,o_{dk_2}})[\wedge o_{dk_2}.val[1] = s] \wedge$	//for any object required by $proc_i$ that exists in the context of the parent process,
(4.24.3) $\neg\exists f, k_3.\ f \in [1..\|Procs\|] \wedge k_3 \in [1..K] \wedge p_{fk_3} \notin p_{ik}.ancestors \wedge (consume_{fk_3,dk_2} \vee consume'_{fk_3,dk_2})$	//if there does not exist another process that is not an ancestor of $proc_i$ and that holds a consumption link with the object in the current configuration,
(4.24.4) $\wedge (type(l) = \text{“}effect\text{”} \rightarrow effect'_{ik,dk_2}) \wedge$	//the object enters an input set of $proc_i$ via an effect link,
(4.24.5) $\wedge (type(l) = \text{“}consume\text{”} \rightarrow consume'_{ik,dk_2})$	//or via a consumption link,

(4.24.6) $\wedge$ ($type(l)$ = "$instrument$" $\rightarrow$ $instrument'_{ik,dk_2}$)	//or via an instrument link.
(4.24.7) $\wedge\, l \in p_{ik}.entries$	//The link is added to the input set of the process $proc_i$.

The following subtransitions are taken after (4.24.0-4.24.7).

(4.25.0) $\forall i, k.\ i \in [1..\lvert Procs\rvert] \wedge k \in [1..K] \wedge p_{ik}.isWaiting \wedge$	//For any waiting process,
(4.25.1) $\wedge\, \forall l.l \in p_{ik}.entries \models \varphi_{entries}(proc_i) \wedge \varphi_{condition}(proc_i) \rightarrow \neg p_{ik}.isWaiting' \wedge p_{ik}.isActive \wedge$	//if the process input set satisfies the precondition of the process, it becomes active.
(4.25.2) $\forall l.\ l \in ProcLinks \wedge type(l)$ = "$in\text{-}zoomed$" $\wedge\ l = (proc_i, proc_j)$	//Invoke all the subprocesses of in-zoomed process $proc_i$,
$\wedge partialOrder(proc_j) = 0 \wedge$	//if they are located at the position along the zero execution time line of $proc_i$.
(4.25.3) $\exists k_1.\ k_1 \in [1..K] \wedge \neg p_{jk_1}.isActive \wedge \neg p_{jk_1}.isWaiting \rightarrow ev\_create\_p_{jk_1}\_p_{ik} \in Ev'$	//Find unused subprocess instances and generate process creation events. This rule is recursive and invokes (4.24.0).

4.5.3 Transition Assertions

In this section we define rules that are followed through every transition.

4.5.3.1 The Unique ID Condition

The following rules are used to prevent a situation in which via different transition rules taken simultaneously. Few object instances are created or few process instances are invoked with the same ID. These rules simplify the definition of the OPM semantics. Transition systems do not use any lookahead, thereby enabling taking into consideration system variable values in the next configuration while resolving the current transition. Different solutions can be used to overcome this limitation in practice. One of them was used in our implementation of the OPM-to-SMV tool.

4.5.3.2 Process instance identifiers uniqueness:

(5.0.0) $\forall i, j, k, k_1, [s].\ i \in [1..\lvert Procs\rvert] \wedge j \in [1..\lvert Procs\rvert/\lvert Objs\rvert] \wedge k, k_1 \in [1..K][\wedge s \in states(obj_j)] \wedge ev\_invoke\_p_{ik} \in D \wedge ev\_invoke\_p_{ik}\_thing_{jk_1[s]} \in D \wedge !(ev\_invoke\_p_{ik} \in Ev' \wedge ev\_invoke\_p_{ik}\_thing_{jk_3[s]} \in Ev')$	//Two different event variables relating to creation of the same process instance cannot be true in the same configuration.

(5.0.1) $\forall i, j, h, k, k_1, k_2, [s], [s_2].\ i \in [1..|Procs|] \wedge j, h \in [1..|Procs|/|Objs|] \wedge k, k_1, k_2 \in [1..K][\wedge s \in states(obj_j)][\wedge s_2 \in states(obj_h)]\wedge ev\_invoke\_p_{ik}\_thing_{jk_1[s]} \in D \wedge ev\_invoke\_p_{ik}\_thing_{hk_2[s_2]} \in D \wedge !(ev\_invoke\_p_{ik}\_thing_{hk_2[s_2]} \in Ev' \wedge ev\_invoke\_p_{ik}\_thing_{jk_3[s]} \in Ev')$

//A process instance cannot be invoked following two different events in the same transition.

4.5.3.3 Object instance identifiers uniqueness:

(5.1.0) $\forall i, j, k, k_1, [s], [s_1].\ i \in [1..|Objs|] \wedge j \in [1..|Procs|] \wedge k, k_1 \in [1..K][\wedge s \in states(obj_i)][\wedge s_1 \in states(obj_j)] \wedge ev\_create\_o_{ik[s]} \in D \wedge ev\_create\_o_{ik[s_1]}\_p_{jk_1} \in D \wedge \neg(ev\_create\_o_{ik[s]} \in Ev' \wedge ev\_create\_o_{ik[s_1]}\_p_{jk_1} \in Ev')$

//An object instance cannot be created environmentally and by a system process in the same transition.

(5.1.1) $\forall i, j, h, k, k_1, k_2, [s], [s_1].\ i \in [1..|Objs|] \wedge j, h \in [1..|Procs|] \wedge k, k_1, k_2 \in [1..K][\wedge s \in states(obj_j)][\wedge s_1 \in states(obj_h)] \wedge ev\_create\_o_{ik[s]}\_p_{jk_1} \in D \wedge ev\_create\_o_{ik[s_1]}\_p_{hk_2} \in D \wedge !(ev\_create\_o_{ik[s]}\_p_{jk_1} \in Ev' \wedge ev\_create\_o_{ik[s_1]}\_p_{hk_2} \in Ev')$

//An object instance cannot be created by two different processes in the same transition.

4.5.3.4 The Links Compatibility

(5.2.0) $ev\_change\_t_a\_p_{ik}\_t_v \in Ev \wedge ev\_change\_t_j\_p_{ik}\_t_v \in Ev \wedge v \in [1..N] \wedge j \in [1..N] \wedge a \in [1..N] \wedge a \neq j \rightarrow sysTermination \in Ev'$	//A process instance cannot be in the same transition in more than one execution timeline.

Chapter 5

OPM-CTS Framework Applicability

The complexity of an OPM model and numerous borderline cases makes the task of evaluation and assessment of the defined OPM operational semantics quite challenging. However, the systematic translation of all the OPM core entities, their features and their interactions into the CTS model helps to cover the OPM core semantics by construction.

The completeness of the translation is only one concern in the process of the evaluation of operational semantics. Another important aspect of the evaluation is the applicability of the introduced operational semantics. In this chapter, I describe a case study project that was carried out to evaluate and assess the operational semantics defined through the work. Through the project, a transition tool from OPM into Symbolic Model Verification(SMV) was developed. The tool provides an implementation of the OPM-to-CTS operational semantics using a standard verification tool, NuSMV [3]. The implementation is partial and it addresses the verification of OPM models that describe in molecular biology processes.

The implementation of the OPM-to-NuSMV translation tool was carried out as a final annual project by a team of five undergraduate students from the Computer Science and Industrial Engineering departments. I have proposed and closely supervised the project. Over a period of one year, we had weekly meetings, during which the system core design, acceptable assumptions regarding the system inputs, technical challenges and their solutions

were discussed. Incrementally, operational rules were incorporated into the translation tool.

During the same period of time, I started working in collaboration with a Ph. D. student, Judith Somekh, on the applicability of the OPM-based verification approach in the field of molecular biology. To make the approach useful for biology researchers, most of whom have lit

In the final stage, the two teams collaborated to carry out the two projects. On the one hand, the front-end application used naming conventions of the OPM-to-NuSMV translation tool to generate SMV specifications. On the other hand, the OPM-to-NuSMV tool was checked using unit tests based on specifications describing the expected dynamic behavior for every single operational rule. The specifications were generated through the front-end application. The tests appeared to be effective, and few bugs in the logic implemented by the OPM-toNuSMV tool were revealed by Chhaya and corrected by the students team.

In Figure 5.1, an OPM diagram describes the normal OPM model verification process that incorporates the OPM-to-NuSMV module. At the beginning, the molecular biology researcher defines an OPM model using the OPCAT tool. Then, the OPM-to-NuSMV plug-in translates the model into SMV script in the NuSMV compatible format. The biology researcher inserts a set of facts into the SMV specification generator (the front-end, a stand-alone tool). The facts are then translated into SMV specifications. Finally, the researcher activates the NuSMV tool with the generated SMV script and SMV specifications and then receives notification on the verification success. In case of failure, a counter example of the system trace including the failure, is shown.

5.1 OPM–to–nuSMV Tool Implementation

5.1.1 Tool Architecture and Inputs

In this section the implementation of the OPM-to-nuSMV is described. The tool was implemented as a plug-in for OPCAT. Via the OPCAT API, the tool receives the OPM model information needed to generate the SMV script. A user may also specify a few configurable module parameters:

- K - the number of instances that can be generated for any OPM object

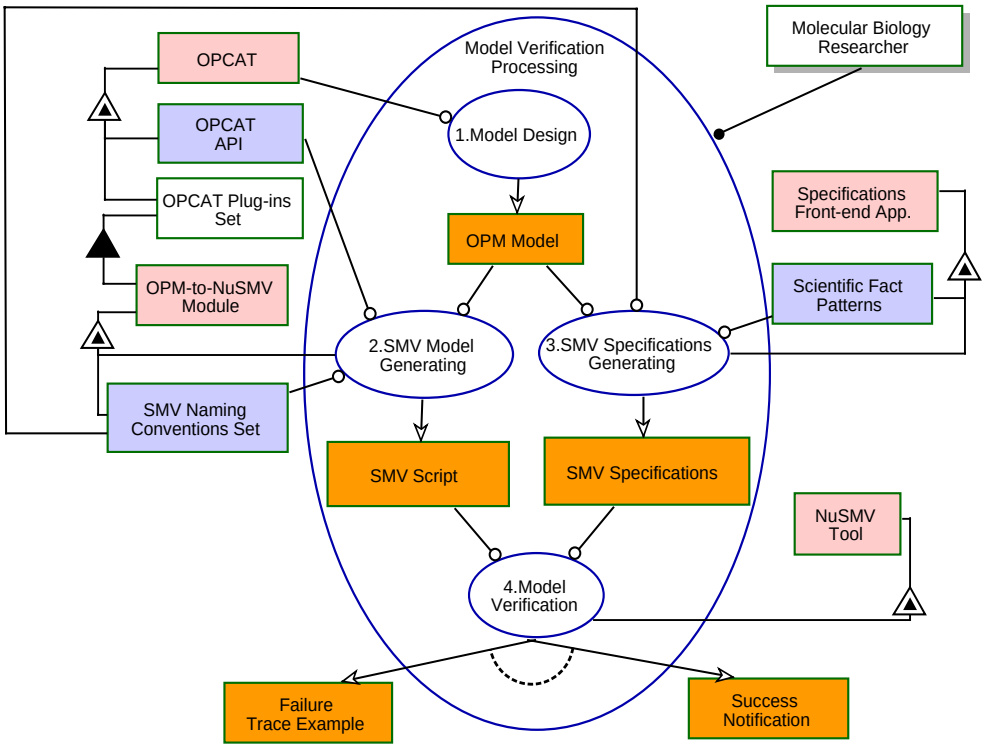


Figure 5.1: OPM Models Verification Process with NuSMV Tool.

or process. The tool was checked using K=2.

- p1 - the probability of generating a process instance termination event during an event transition.
- p2 - the probability of generating a system termination event during an event transition.

The tool is activated on the currently open OPM model in OPCAT. The model must satisfy the following conditions:

- The model may only include those OPM mechanisms that were defined in the work of Judith Somekh as a sufficient subset of the OPM language to express systems in molecular biology. In particular, the links treated by the translation module are: aggregation-participation,

exhibition-characterization, classification-instantiation, bi-directional association, result, consumption, effect, condition, instrument and invocation.

- The module does not provide a solution for mutual exclusions between two or more different processes that can generate the same object instance simultaneously.
- OR/XOR gates are possible only between result links in the current implementation.

5.1.2 Logic of the SMV System

In Figure 5.2 describes all the system states and the transitions between them.

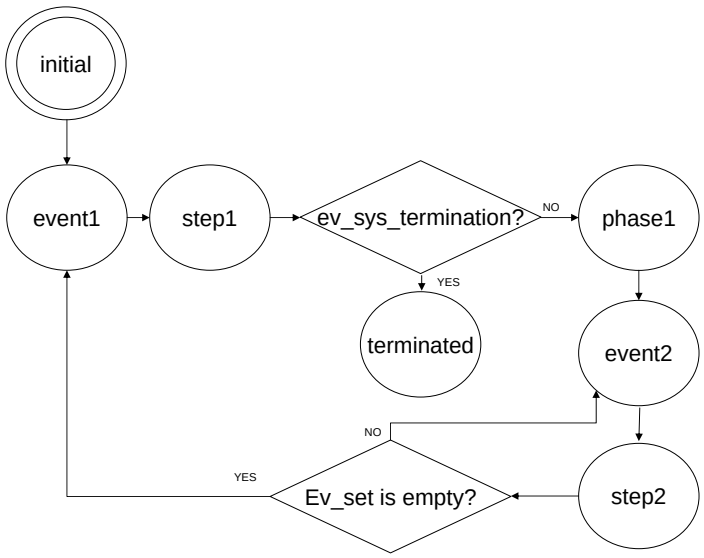


Figure 5.2: The OPM/SMV transition diagram.

The transition system includes initial and final states (the states labeled "initial" and "terminated") and two loops. The initial state describes the initial settings of all the system variables representing the OPM process and object instances. The "terminated" state is implemented as a Boolean flag set to the true value. Once the flag becomes true, the system cannot progress any further using any existing transitions.

The external loop includes all the states except for the initial state. Its precondition is checked directly after finishing the transitions related to the step2 state. If a set of events that can be treated in the system via a set of deterministic transitions is empty, the precondition is satisfied and the system moves to the event1 state. The event1 state is used to simulate non-deterministic system inputs and the next iteration of the external loop starts. Another loop includes only two states: event2 and step2. The loop is taken until the event set (all the variables that represent system events that trigger an object creation, an object consumption, a system termination and others) is empty.

The event1 state includes transitions where the system randomly picks states of environmental objects and determines which active processes shall be terminated. In addition, during this state, a system termination event may be generated.

The step1 state handles the mutual exclusion between instances belonging to the same object which are generated by different instances of the same process. In addition, within this state, unused object instances are detected for all the objects to be created in the following transitions.

Finally, the step2 and the event2 states include the set of all the deterministic operations triggered by a set of events generated within the previous steps. The step2 state also includes transitions which create new association and participation links between newly created objects and other objects.

5.1.3 Examples of OPM-to-NuSMV Translation Rules

In this section we provide several translation rules and demonstrate them using a simple diagram and the appropriate automatically-generated code.

1. Following is an example of how an event of type $ev\_create\_processName\_objectName$ within the step1 state is treated. The event triggers the creation of an

object instance upon a process termination event if the process holds a result link to the object.

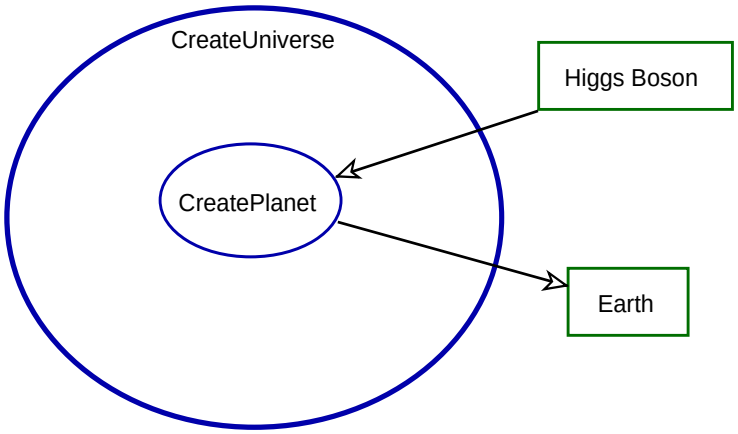


Figure 5.3: OPM object creation and consumption.

In Figure 5.3, an object instance of **Earth** is created upon the **CreatePlanets** process instance termination following the result links connecting the object with the process. The following SMV lines were generated by the OPM-to-NuSMV tool we have developed.

```
1. next(ev_create_CreatePlanets_0_Earth) := case
2. sysState = step1 & ev_terminate_CreatePlanets_0=0 &
!(earth_0 = notExist) & !(earth_1 = notExist) : false;
3. sysState = step1 & ev_terminate_CreatePlanets_0=0 &
!(earth_1 = notExist): 0;
4. sysState = step1 & ev_terminate_CreatePlanets_0=0 &
!(earth_1 = notExist): 1;
5. sysState = step1 & ev_terminate_CreatePlanets_0=0 &
!(earth_1 = notExist): {0, 1};
6. sysState = event1 & ev_create_CreatePlanets_0_earth = 0 &
earth_0=notExist : false;
```

```
7. sysState = event1 & ev_create_CreatePlanets_0_earth = 1 &
earth_1=notExist : false;
8. TRUE : ev_create_CreatePlanets_0_earth;
9. esac;
```

Description:

- Line 1 - If two instances of the Earth object are not in the state *notExist*, then the object instance should not be created. The event *ev_create_CreatePlanets_0_earth* won't become true within the step1 state.
- Lines 2-3 - the condition in line 1 was false. Hence, there exists at least one instance belonging to the object *Earth* that can be used. Each one of the lines treats a case when only one object instance is free and can be used.
- Line 4 - there are no existing object instances. Thus, both instances with index zero and with index 1 can be used. Then, the object instance index is picked randomly.
- Lines 5-6 - if the system is in the event1 state, the event has been treated and must be set to false.

Were the object **Earth** to hold two states, **liquid** and **solid**, and the process **CreatePlanets** be connected to the **liquid** state via a result link, then the following SMV lines would be added to the script and belong to the event2 state:

```
1. next(ev_create_CreatePlanets_0_earth_0_liquid) := case
2. sysState = event2 & ev_terminate_CreatePlanets_0 = 0 &
earth_0 = notExist & ev_create_CreatePlanets_0_earth=0 : TRUE;
3. sysState = event2 : FALSE;
4. TRUE : ev_create_CreatePlanets_0_earth_0_liquid;
5. esac;
```

Description:

- Line 2 checks that the current state is event2, the process that creates a new object is in the process of termination, and an instance with zero index is selected for the new object instance.
- Line 3 sets the event variable to false.

2. Following is an example of how environmental object creation and object consumption events are treated. In Figure 5.3, an object instance of **Higgs Boson** is created as an environmental object and the process **CreatePlanets** consumes it. The following SMV lines will be added to the step1 and step2 states to describe the object creation and its consumption.

```
1. next(higgs_boson_0) := case
2. sysState = step1 & higgs_boson_0 = notExist &
ev_create_higgs_boson_0 = 0 :exist;
3. sysState = step2 &
ev_consumption_higgs_boson_0_CreatePlanets_0 = TRUE : notExist;
4. sysState = step2 &
ev_consumption_higgs_boson_0_CreatePlanets_1 = TRUE : notExist;
5. TRUE : higgs_boson_0;
6. esac;
```

Description:

- Line 2 refers to the creation of an environmental object occurring during the step1 state;
- Lines 3-4 show the consumption of the object.

5.2 OPM-to-NuSMV Tool Testing

The tool was tested on a model that describes a real molecular process, called the Transcription Cycle. The model was prepared by Judith Somekh as part of her research. It includes approximately 35 diagrams. The opz file (OPM format) size is about 52KB and the output script for K=2 is about 500KB. The output SMV script passed the NuSMV compilation successfully.

In addition, the tool passed a set of unit tests based on specially-designed SMV specifications describing basic OPM operational rules. The rules included OR/XOR gates, the creation of a new object, the creation of an object in a specific state, and the movement among subprocesses within an in-zoomed process. All these tests passed successfully.

Chapter 6

Summary and Future Work

Model-based systems engineering approaches are becoming accepted in modern industry. The approaches are focused on system models, which are the main target within the initial development stages. System conceptualization is one of the critical initial stages. During this stage, a multidisciplinary team, including all the system engineering stakeholders, defines the main concepts and boundaries of the future system. This stage highly depends on characteristics of the modeling language and methodology selected to define the system. High expressiveness, which enables definition of software-aided and hybrid systems, exact semantics and usability are three desired characteristics making a conceptual modeling language appropriate for the conceptual modeling stage.

Object-Process Methodology (OPM) is a holistic, integrated approach to information and systems engineering. Usability and expressiveness of the methodology has been studied through numerous case-studies from disparate areas. While being simple, hierarchical, and highly expressive, the OPM modeling language has lacked formal operational semantic definition.

In this work, the problem related to the lack of formal definitions of OPM as a modeling language has been addressed. The following are the main contributions of the work:

1. defining OPM operational semantics via a formal computational model, enabling a formal process for further language extensions,

2. developing the OPM semantics for numerous process and object instances, while accounting for structural and procedural links among

the instances,

3. detecting of inconsistent OPM operational rules through formalization of the language and redefinition of these rules, and

4. verifying the applicability of the rules by construction of the OPM model-based verification tool based on the defined operational semantics.

There exist similar studies in the formal definition of operational semantics for modeling languages. Typically, the goal of the works is to define model verification tools using operational semantics. To study the applicability of the defined rules we have also implemented a model-based verification tool. The major focus of the work was to analyze and to enhance OPM operational semantics using a formalization process. This task was particularly challenging due to the richness of the OPM language. Our focus was to define the operational semantic rules. This included rules to manage object and process instances. This type of rules does not exist in definitions of most other works focusing on model-based verification.

The definition of the OPM operational semantics was implemented through the construction of translation from the OPM language into a clocked transition system (CTS) formalism. The translation included the following definitions:

1. definition phases comprising the OPM system life cycle, which include init, tick, event, and step transitions,

2. definition of variables that represent the OPM object and process instances with procedural and structural links among them,

3. definition of input variables simulating the OPM events and nondeterministic selections, and

4. definition of sets of transitions describing OPM system state changes which are selected depending on the current phase in the system life cycle.

To explore the applicability of the defined OPM operational semantics, an OPM-to-SMV translation tool has been developed as part of the research. The translator accepts as input an OPM model and automatically generates

a script in the SMV verification language. The translator has become an important part in the research methodology proposed by Judith Somekh in the field of molecular biology. Following this method, information found in the field of molecular biology will be translated into a holistic model in OPM by patterns defined by Judith in her work.

In our collaborative work, based on the OPM model, we propose, using of the OPM-to-SMV translation tool for model verification. According to this approach, the biologist will be able to verify the model by running an SMV file created automatically from the model using the tool. The model will be checked against specifications which the investigator inserts into the NuSMV tool. The specifications will be derived from the biological facts found in published research papers from which the OPM model is also constructed.

The usability of a tool like NuSMV by biologists is questionable, since most of them have no appropriate background in formal verification required to work with such tools. We made an effort to provide the researcher with a graphical, friendly front-end, through which he/she can insert biological facts into NuSMV without the need to understand either the generated SMV script or the SMV specification semantics.

The result of running the SMV script on NuSMV is successful if the SMV model, defined by the script, satisfies all the inserted specifications, or an example of an SMV trace that terminates with an invalid system state. The trace relates to the state variables that appear in the input SMV model rather than to the OPM entities. Since in our translator the implementation SMV variable names are close to the original OPM entity names that the variables represent, it is relatively easy to understand the relation between the SMV trace and the original OPM model.

Possible extensions to this work are the following:

- Extension of the OPM operational semantics by OPM advanced mechanisms that were excluded in the current definition, such as loop and recursion mechanisms.
- Construction of a model-based simulator implementing the rules defined in this work to enable qualitative analysis of an OPM model.
- Translation of NuSMV traces shown as system failure examples into OPM traces. The traces can be then played using OPCAT animation module.

- Explore a method to examine hypotheses in the field of molecular biology using an OPM model and model verification capabilities. The researcher will use the biological model that he/she has already created and that successfully passed validation process and extend the model with new hypotheses. Then the investigator should translate the model into SMV and verify the resulting SMV script with the specifications that include all the biological facts used to verify the model before the hypotheses were added. Using this method, the researcher may be able to find logical inconsistencies in his/her hypothesis even before starting wet laboratory experiments. Discovering problems in hypothesis in that early stage will save months of research, resources and money.

To summarize, this work was devoted to an in-depth study of the operational semantics of OPM language, resolving inconsistencies between operational rules, offering enhanced rules, and finally providing formal OPM operational semantics. The formal definition of the semantics enables introducing new language structures in a more formal process with higher attention to possible inconsistencies caused by the new constructs. The formal process of inserting new OPM constructs will include adding appropriate translation rules of the new constructs into a CTS model representing the equivalent of a given OPM model. The OPM formalization process itself played an important role in revealing inconsistencies among OPM rules. Finally, the formal definition of OPM rules will enable development of OPM model test methods including model simulation, formal verification and animation, with all of these testing methods founded on a formal uniform set of rules.

Appendix

The following three tables describe all the OPM native operators. The two first tables include all the built-in arithmetic and Boolean operators, such as multiplying, dividing, and comparing. Note that the operators are defined only for OPM Numeric, and in some cases Time, Date and String types or any other type that is compatible with these types.

The third table describes two advanced operators that work on OPM compound data structures. The operator '[]' works on OPM containers: Array, Set and Map. It takes as an argument a positive value of a type compatible with integer and returns an object associated with this value from the Array or from the Set. It can also take an OPM object **B** as its argument and find an object kept in the Map and associated with the key. The second special operator is an identity operator, which is used to refer to an object value or to one of its attributes.

All the tables hold four columns: the symbol that stands for the operator, the operand types, the result type, and the name of the operator. All the built-in operators from the first two tables have their typical semantics (similar to the one employed by Java).

Table 1: Native OPM processes invoked on objects of types A, B

Op	**Operand Types**	**Result Type**	**Operation**
$*$	A, B both $\preceq Numeric$	$max(A, B)$	Multiplying
$/$	A, B both $\preceq Numeric$	$max(A, B)$	Dividing
$\%$	A, B both $\preceq Numeric$	$max(A, B)$	Remainder Calculating
$+$	A, B both $\preceq Numeric$	$max(A, B)$	Adding
	$A \preceq Date$, $B \preceq Time$	$Date$	
	$A \preceq Time$, $B \preceq Time$	$Time$	
	A, B both $\preceq String$	$String$	Concatenating
$-$	A, B both $\preceq Numeric$	$max(A, B)$	Subtracting
	$A \preceq Date$, $B \preceq Time$	$Date$	
	$A \preceq Time$, $B \preceq Time$	$Time$	
$<$	A, B both $\preceq Numeric$	$Boolean$	IsSmaller
	A, B both $\preceq String$		
	A, B both $\preceq Date$		
	A, B both $\preceq Time$		
$>$	A, B both $\preceq Numeric$	$Boolean$	IsBigger
	A, B both $\preceq String$		
	A, B both $\preceq Date$		
	A, B both $\preceq Time$		

Table 2: Native OPM processes invoked on objects of types A, B

Op	**Operand Types**	**Result Type**	**Operation**
$\leq$	A, B both $\preceq Numeric$ A, B both $\preceq String$ A, B both $\preceq Date$ A, B both $\preceq Time$	$Boolean$	IsSmaller or IsEqual
$\geq$	A, B both $\preceq Numeric$ A, B both $\preceq String$ A, B both $\preceq Date$ A, B both $\preceq Time$	$Boolean$	IsBigger or IsEqual
$=$	A and B, such that $B \preceq A$, and optional $C \in [1..5]$ for informational objects	A	Assignment
$==$	A and B, such that $A \preceq B$ or $B \preceq A$, and optional $C \in [1..5]$ for informational objects	$Boolean$	IsEqual

Table 3: Object Reference Operators

Op	**Operand Types**	**Result Type**	**Operation**
$[i]$	$Array\langle A\rangle$ or $Set\langle A\rangle$, $i \in B$, such that $B \preceq Integer$ $Map\langle A, B\rangle, i \in B$, such that A and B are OPM Objects (B denotes the map key type)	A	Element at cell i
@	OPM Object	Object Value	Identity

Bibliography

[1] Opcat - homepage.

[2] iumlite - homepage, 2007.

[3] Nusmv: a new symbolic model checker, 2011.

[4] puml - homepage, 2011.

[5] Uml action semantics, 2011.

[6] Rajeev Alur and David L. Dill. The theory of timed automata. In *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, pages 45–73, 1991.

[7] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *REX Workshop*, pages 74–106, 1991.

[8] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993.

[9] Modelica Association. A unified object-oriented language for physical systems modeling: Language specification version 3.0, September 2007.

[10] Arieh Bibliowicz. *A Graph Grammar-Based Specification of Object-Process Methodology.* 2008.

[11] Nikolaj Bjørner, Zohar Manna, Henny Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using step. *Theor. Comput. Sci.*, 253(1):27–60, 2001.

[12] Muffy Calder and Carron Shankland. A symbolic semantics and bisimulation for full lotos. In *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XIV*, pages 184–200. Kluwer Academic Publishers, 2000.

[13] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[14] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[15] Dov Dori. Object-process methodology as a business-process modelling tool. In *ECIS*, 2000.

[16] Dov Dori. *Object-Process Methodology, A Holistic Systems Paradigm.* Springer, 2002.

[17] Rik Eshuis, David N. Jansen, and Roel Wieringa. Requirements-level semantics and model checking of object-oriented statecharts. *Requir. Eng.*, 7(4):243–263, 2002.

[18] Rik Eshuis and Roel Wieringa. Tool support for verifying uml activity diagrams. *IEEE Trans. Software Eng.*, 30(7):437–447, 2004.

[19] Günter Graw and Peter Herrmann. Transformation and verification of executable uml models. *Electr. Notes Theor. Comput. Sci.*, 101:3–24, 2004.

[20] Y. Grobshtein and D. Dori. Generating sysml views from an opm model: Design and evaluation. *Systems Engineering*, 14:327–340, 2011.

[21] Yariv Grobshtein, Valeria Perelman, Eli Safra, and Dov Dori. Systems modeling languages: Opm versus sysml. In *Systems Engineering and Modeling, ICSEM'07*, June 2007.

[22] Masami Hagiya and John C. Mitchell, editors. *Theoretical Aspects of Computer Software, International Conference TACS '94,*

Sendai, Japan, April 19-22, 1994, Proceedings, volume 789 of *Lecture Notes in Computer Science*. Springer, 1994.

[23] David Harel, Amir Pnueli, and Jonathan Stavi. Propositional dynamic logic of nonregular programs. *J. Comput. Syst. Sci.*, 26(2):222–243, 1983.

[24] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for timed transition systems. *Inf. Comput.*, 112(2):273–337, 1994.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] Joseph Hofstader. Model-driven development, 2011.

[27] Alan J. Hu, Masahiro Fujita, and Chris Wilson. Formal verification of the hal s1 system cache coherence protocol. In *ICCD*, pages 438–444, 1997.

[28] Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.*, 12(9):890–904, 1986.

[29] Inc. Kabira Technologies. Inc. kabira action semantics.

[30] Rick Kazman, Rick Kazman, Mark Klein, Mark Klein, Paul Clements, Paul Clements, Norton L. Compton, and Lt Col. Atam: Method for architecture evaluation, 2000.

[31] Yonit Kesten, Zohar Manna, and Amir Pnueli. Verification of clocked and hybrid systems. *Acta Inf.*, 36(11):837–912, 2000.

[32] Philippe Kruchten, Rich Hilliard, Rick Kazman, Wojtek Kozaczynski, J. Henk Obbink, and Alexander Ran. Workshop on methods and techniques for softwaer architecture review and assessment (sara). In *ICSE*, page 675, 2002.

[33] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[34] Juncao Li, Fei Xie, Thomas Ball, and Vladimir Levin. Efficient reachability analysis of büchi pushdown systems for hardware/software co-verification. In *CAV*, pages 339–353, 2010.

[35] Orna Lichtenstein and Amir Pnueli. Propositional temporal logics: Decidability and completeness. *Logic Journal of the IGPL*, 8(1), 2000.

[36] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. Statecharts via process algebra. In *CONCUR*, pages 399–414, 1999.

[37] Zohar Manna and Amir Pnueli. Models for reactivity. *Acta Inf.*, 30(7):609–678, 1993.

[38] John D. McCharen, Ross A. Overbeek, and Larry Wos. Problems and experiments for and with automated theorem-proving programs. *IEEE Trans. Computers*, 25(8):773–782, 1976.

[39] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[40] Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360, 1997.

[41] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. pages 526–548. Springer-Verlag, 1991.

[42] OMG. Omg systems modeling language (omg sysml), version 1.2, 2010.

[43] Object Management Group (OMG). Unified modeling language: Infrastructure, version 2.0.

[44] Object Management Group (OMG). Unified modeling language: Superstructure, version 2.0.

[45] Object Management Group OMG. *Meta Object Facility (MOF)*. 2000.

[46] Jonathan S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, 1992.

[47] Mor Peleg. *Modeling Systems Dynamics Through The Object-Process Methodology*. 1999.

[48] Mor Peleg and Dov Dori. Extending the object-process methodology to handle real-time systems. *JOOP*, 11(8):53–58, 1999.

[49] Mor Peleg and Dov Dori. The model multiplicity problem: Experimenting with real-time specification methods. *IEEE Trans. Software Eng.*, 26(8):742–759, 2000.

[50] Mor Peleg, Judith Somekh, and Dov Dori. A methodology for eliciting and modeling exceptions. *Journal of Biomedical Informatics*, 42(4):736–747, 2009.

[51] Valeria Perelman, Judith Somekh, and Dov Dori. Clocked transition system as an opm formalizm with application to systems biology. In *Simulation Multiconference, SpringSim'11*, April 2011.

[52] Gergely Pintér and István Majzik. Runtime verification of statechart implementations. In *WADS*, pages 148–172, 2004.

[53] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. 1986.

[54] Inc. Project Technology. Bridgepoint action language (al) manual.

[55] Iris Reinhartz-Berger, Dov Dori, and Shmuel Katz. Developing web applications with opm/web. In *DIWeb*, pages 47–61, 2001.

[56] Jan Romberg, Oscar Slotosch, and Gabor Hahn. Mode: A method for system-level architecture evaluation. In *MEMOCODE*, pages 13–23, 2003.

[57] John M. Rushby and Friedrich W. von Henke. Formal verification of algorithms for critical systems. *IEEE Trans. Software Eng.*, 19(1):13–23, 1993.

[58] Wladimir Schamai. Modelica modeling language (modelicaml): A uml profile for modelica. Technical report, 2009.

[59] Rick Kazman Sei, Len Bass, Linda Northrop Sei, Gregory Abowd, Paul Clements Sei, Paul Clements Sei, Amy Zaremski Sei, and Amy Zaremski Sei. Recommended best industrial practice for software architecture evaluation, 1997.

[60] Robert E. Shostak, editor. *7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings*, volume 170 of *Lecture Notes in Computer Science*. Springer, 1984.

[61] Graeme Smith and Ian J. Hayes. An introduction to real-time object-z. *Formal Asp. Comput.*, 13(2):128–141, 2002.

[62] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[63] Arnon Sturm, Dov Dori, and Onn Shehory. Engineering mobile agents. In *ICEIS (4)*, pages 79–84, 2008.

[64] Jorn Guy Su? and Adrian Pop. The impreciseness of uml and implications for modelicaml. pages 17–26, 2008.

[65] Masahito Sugai, Akira Teruya, Eiichiro Iwata, Nurul Azma Zakaria, Noriko Matsumoto, and Norihiko Yoshida. Assertion-based dynamic verification for executable uml specifications. In *Proceedings of the 8th conference on Applied computer scince*, pages 181–186, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[66] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *CONCUR*, pages 2–17, 1994.

[67] Michael von der Beeck. A comparison of statecharts variants. In *FTRTFT*, pages 128–148, 1994.

[68] Ian Wilkie, Adrian King, and Mike Clarke. Uml asl reference guide, 2011.

[69] Kirsten Winter and Graeme Smith. Compositional verification for object-z. In *ZB*, pages 280–299, 2003.

[70] Mikai Yang, Greg J. Michaelson, and Rob Pooley. Formal action semantics for a uml action language. *J. UCS*, 14(21):3608–3624, 2008.

[71] Yevgeny Yaroker, Valeria Perelman, and Dov Dori. Opm model based simulation environment for systems engineering conceptualization phase. In *Model-Based Systems Engineering, MBSE'09*, March 2009.