Thèse

présentée à

L'Université Pierre & Marie Curie – Paris VI

en vue de l'obtention du titre de

Docteur de l'Université Pierre et Marie Curie

Spécialité :

Systèmes Informatique

par

Reda BENDRAOU

Sujet de la thèse :

UML4SPM: Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle

A soutenir le 6 septembre 2007 devant un jury composé de :

Jean-Marc JEZEQUEL	Rapporteur	Professeur à l'Université Rennes 1
Pierre-Alain MULLER	Rapporteur	MdC. HDR à l'Université de Mulhouse
Colin ATKINSON	Examinateur	Professeur à l'Université de Mannheim
Bernard COULETTE	Examinateur	Professeur à l'Université de Toulouse
Fabrice KORDON	Examinateur	Professeur à l'Université Paris VI
Marie-Pierre GERVAIS	Directeur	Professeur à l'Université Paris X
Xavier BLANC	Co-encadrant	Maître de Conférences Paris VI

Table of contents

Chapter 1

Introduction

Motivation

Building complex and reliable software systems in the shortest time-to-market represents the challenging objective that competitive companies are facing everyday. Projects developing large computer software may span months and involve many development teams, working on different locations and using diverse tools and technologies. All these project's variants make it very difficult for companies to respect product delivery deadlines and estimated costs. A more challenging objective for these companies is to be able to repeat the same development project within the same deadlines and to provide software with the same quality. According to the Standish Group report, in 2006, only 35% of software projects were completed on time, on budget and met user requirements, while 46% had cost or time overruns or didn't fully meet the user's needs and 19% have failed (cancelled prior to completion or delivered and never used) [Standish 06].

In the software industry, these objectives are not new. What is constantly changing, is the *technologies* and *ways of working* used to reach these objectives. By *technologies*, we refer to the set of programming and modeling languages, methodologies, applications, middleware, systems and platforms, etc., used for building software. As for *ways of working*, it regroups the set of techniques, best practices, and strategies followed in order to effectively manage the development efforts of large teams of software engineers over the software construction's phases.

Over the past several years, the "*ways of working*" dimension has gained a particular attention in the software industry. This is due to the fact that since late eighties, there has been a growing conviction that software systems should be viewed as products resulting from the execution of orderly *software development processes* [Sutton 95a] [Montangero 99]. Consequently, the quality of a software product cannot be ensured simply by inspecting the product itself or by performing the traditional verification and validation approaches (V&V) [ANSI/IEEE 87]. It also relates to the production process that is carried out and to actors involved in this production process. There is, further, a belief that software development times and costs can be reduced, and software product quality can be improved through the disciplined application of superior *software processes*. These beliefs have served to focus attention on the problem of *how to create and to represent superior software processes* [Sutton 95b].

Lonchamp defines a software process as "*the set of partially ordered process steps, with sets of related artifacts, human and computerized resources, organizational structures and constraints, intended to produce and maintain the requested software deliverables"* [Lonchamp 93]. Looking at this definition, we can notice the important number of factors that may affect the success of software development processes. Indeed, software processes are not typical production processes. They are complex and highly unpredictable since they depend too much on too many people and circumstances. Not all activities of the software process require automation and depend on communication, coordination and cooperation within a predefined framework [Ruiz 04].

Thus, the real challenge of software development firms is to find the means of rationally *representing* and *managing* activities, resources and constraints of their software development processes while taking into account all these characteristics. Once these processes represented and capitalized in *process models*, they become an important asset of the company. Process models can then be used to reason about processes, to test them and to improve them in order to answer to quality and cost expectations.

Hosier [Hosier 61], Royce [Royce 70] and Boehm [Boehm 76] were among the pioneers to propose models of the Software Development Life Cycle (SDLC). However, the software process community was unsatisfied with these models. The granularity of process steps included in SLDC models was too large and does not prescribe the precise course of actions, artifacts and tools used within the process, development policies and constraints, etc [Curtis 92].

Rapidly, the need to describe in more details processes that software companies are actually performing during software development or maintenance emerged. Consequently, the software domain saw the spreading of a multitude of *Software Process Modeling Languages (SPMLs)*. Some of them were rules-based (e.g., MARVEL) [Kaiser 90], others Petri-net based (e.g., SPADE) [Bandinelli 93] or programming languages based (e.g., SPELL, APPL/A) [Conradi 92] [Sutton 95]. They are commonly called first-generation languages.

However, no language has gained general acceptance or widespread. Except the fact that these languages were executable, they had obvious limitations. They were based on existing paradigms that were not particularly well adapted to the domain of software process modeling [Sutton 97]. Their complexity, their use of low-level formalisms and the impossibility for non-programmers to use and to understand them, were among the obstacles for their adoption. The large number of process stakeholders with different backgrounds ranging from software process engineers, project managers, system engineers to customers imposes the use of an understandable and intuitive representation format.

In front of the limited success of first-generation SPMLs, other languages promoting simplicity by using high-level constructs, narrative text and graphical representations have been proposed. Examples of such process descriptive languages are the OMG's (Open Management Group) initiative for software process modeling called, SPEM (Software Process Engineering Metamodel) [OMG 02] [OMG 07c] and the SEMDM (Software Engineering — Metamodel for Development Methodologies) specification, a standardisation effort initiated by ISO [ISO 06]. Contrarily to first-generation languages, the practical utility of these languages is limited to documenting methodologies and processes. They are rather suitable for enterprise's expertise and knowledge capitalization, process description exchanges than to be executed. Documented processes can help in the comprehension of the process however; they cannot be used for automating some process routings and for coordinating between activities and software engineers of the development process. For that latter purpose, advanced constructs are required to capture control and data flows, iterations, choices, exceptions, communication between agents, tool invocations, and so on. Nowadays, companies are looking for how to extensively automate all parts participating in the software production, including the development process itself.

Of course, software processes cannot be fully automated since they are human-centred processes. However all steps and coordination controls that do not require human interventions can be automated (e.g. starting activities, routing of artifacts across process's activities, handling of exception, deadlines and alarms management, etc.).

With the popularization of the OMG's Unified Modeling Language (UML) and surfing on the MDA (Model Driven Architecture) wave [OMG 03], the process modeling community saw in UML, a potential candidate as a SPML. UML is standard, provides high-level constructs, offers a rich notation and a set of diagrams and is wide spread. Thus, many propositions emerged. More cited ones are Promenade [Franch 98], Di Nitto's approach [Di Nitto 02] and Chou's approach [Chou 02]. The common lack between these languages is that they neither do propose a new language nor extends the UML language. Language's elements (grammar) come in form of a UML class diagram. Thus, when the process modeler needs to define its process model she/he has to extend the predefined UML class diagram provided by the approach (e.g. make all process's activities as a specialization of the class *Activity* in the predefined UML class diagram). However, there is no proper semantic for these elements and no appropriate notation. They all have the same semantics as the UML *Class* metaclass since they are all instances of this metaclass. Almost all basic process elements are given and are represented as a UML class diagram.

Regarding executability, in Chou's approach, the UML class diagram is used only to reason about the process. In order to execute the process model, the process modeler has to code manually the process in a proprietary programming language. In Di Nitto's approach, the process program is generated from the multiple UML diagrams used to define it (i.e., class, activity and state diagrams). However, process modeler has to add code manually into the program since there are no relationships between the input diagrams. Another limit of this approach is that the code generation is based on name matching of process elements scattered on different input diagrams (i.e. assuming for instance that the process element named "A" in the activity diagram, is the same process element "A" in the class diagram). Finally, the Promenade approach does not provide solutions for the executability requirement.

If we look back to all efforts taken in the area of software process modeling for defining the appropriate SPML, we can clearly distinguish two families of software process modeling languages. *Executable SPMLs*, based on programming languages, and *Descriptive SPMLs* providing high-level construct and graphical representations. However, no software process modeling language succeeded in satisfying these two apparently conflicting requirements. Thus, a trade-off between high-level constructs and executability is needed in order to satisfy the expectations of the software process modeling community.

Another issue that current and first-generation software process modeling languages overlooked is the **human dimension**. Software development processes are all about human creative tasks. The emphasis of first-generation SPMLs has been on describing software process models as *normative* models i.e., on prescribing the expected sequence of activities and pushing automation to enforce them [Cugola 98a]. Humans have a central role in performing the activities needed to accomplish the process goals. They do so by interacting and cooperating among themselves and with computerized tools, by making decisions and by reorienting the current process workflow. Thus, this dimension has to be taken into account while designing a SPML.

The last important issue relates to **flexibility**. According to W. Humphrey, "The actual process is what you do, with all its omissions, mistakes, and oversights. The official process is what books say what you are supposed to do". This quotation, represents quite well implicit challenges that software development firms have to face. The first one is to be able to adapt a software process model to company's specific projects and culture. By adapting, we mean the possibility to customize the process model and to be able to extend it with project-specific components i.e. adding specific artifacts, roles, methodology steps, etc. The second challenge is to not constrain people to follow a predefined pattern of activities, but to provide support to their creative tasks. Software processes are too complex and intrinsically dynamic to be defined in all details in advance. Moreover, no matter how carefully the process is defined, in practice people often deviate from the normative description embodied into the process model [Madhavji 93] [Cugola 98b].

Research objectives

In order to overcome the above-mentioned issues, in this document, we present UML4SPM, a UML2.0-Based Language for Software Process Modeling. *Expressiveness*, *Understandability*, and *Executability* were our main objectives while designing UML4SPM. Our contribution comes in form of MOF-Compliant metamodel, which extends the UML2.0 Superstructure standard [OMG 07b], a simple yet expressive graphical notation and high level constructs with precise execution semantics. UML4SPM process models can be executed directly, without any intermediate or refinement steps. The UML4SPM proposition came as the result of the following research objectives we identified at the beginning and during this work:

- **A large study of the state of the art on the *Process Modeling* domain**. Indeed, process modeling is a very mature field and encompasses many sub domains. The literature is very rich and it becomes very hard to distinguish between the multiple proposed definitions, acronyms and concepts. This objective has in fact many sub objectives.

  The first one is to put in place some definitions that we will use along this work. Examples of such definitions are *Process*, *Process Model*, *Process Modeling Language*, etc. As we will see, the term *Process* for instance may have slightly different meaning form one process modeling community into another which may be confusing.

  The second sub objective is to identify major requirements to take into account while designing a software process modeling language. These requirements were established by several well-known works proposed in the literature. We will use them as means for comparing between the different SPML propositions and as design goals for defining UML4SPM.

  The last sub objective is to clarify the relationship between the different process technology domains. In the recent past, the process modeling domain saw the emergence of several process communities, each one having its own vision and expectations of modeling and executing processes. Examples of such communities are the Workflow Management community (WfM), the Business Process Modeling community (BPM), the Software Process Modeling, (SPM) community, the Enterprise Application Integration (EAI) community, etc. This situation led that nowadays, many process analysts and company deciders are confused about which technology to use to achieve their goals. Besides, they are unable to establish the relationship between these domains. To clarify this situation and for comparing these process technology domains, we defined a *framework* [Bendraou 07b]. This framework gives process definitions, characteristics, modeling objectives, process model constituents, process context and scope proper to each domain. The main goal behind this framework is to guide process modelers and deciders in their choice of the appropriate technology in regard with their process modeling objectives. Conversely, it can be used in order to identify the set of concepts and requirements that have to be respected in order to define a new PML according to the domain.

- **Exploring how the software process modeling community can take advantage of the MDE (Model Driven Engineering) vision**. MDE promises a better software productivity and more reliability at reduced costs. The MDE counts on the use of platforms-independent modeling languages instead of code during software development projects in order to hold these promises. Company's expertise and knowledge are capitalized in models which prevent them from the continuous evolution of technologies and platforms. Productivity and reliability are reached by automating code generation from these models into the appropriate platforms. For the choice of the modeling language to use, the MDE vision does not advocate the use of any specific platform independent modeling language. However, UML succeeded by far to be one of the most popular. The software process community recognized the benefits of the MDE vision and was attracted in applying the MDE principles to the area of software processes. Thus, exploring the suitability of UML as SPML came as natural initiative and since its earlier versions, approaches on defining UML-Based SPMLs were proposed [Franch 98] [OMG 02][Di Nitto 02] [Chou 02]. In this work, we highlight the MDE principles that may influence the software process modeling community in terms of abstraction and productivity and we identify advantages that UML offers as a SPML but also its limits [Bendraou 05a]. We also compare all UML-Based approaches for software process modeling according to the SPML requirement we identified in fulfilling the first objective of this work.

- **Defining the UML4SPM metamodel**. Instead of starting from scratch, we decided to reuse the expressiveness of UML2.0 *Activity* and *Action* packages. In this new version of the UML standard, these packages offer concepts and features that allow the modeling of sophisticated activities and actions with executable semantics. Thus, our first step is to identify the UML2.0 subset of *Activity* and *Action* elements suitable for process modeling. This subset is then used as a basis for defining the UML4SPM metamodel. UML4SPM comes in form of a MOF-compliant metamodel which extends a subset of the UML2.0 Superstructure standard [Bendraou 05a]. UML2.0 provides all the concepts related to the sequencing of activities and actions, for expressing choices, concurrency, synchronization, events, exception handling, etc., while UML4SPM metaclasses provide the semantics proper to software process modeling concepts. We also provide a UML4SPM Process Model Editor based on the Eclipse open source development environment [Eclipse].

  Regarding the UML4SPM notation, it is principally inspired from the UML2.0 *Activity Diagrams* notation. This notation is enriched to take into account some element's important features (properties) for software process modeling purposes.

For UML2.0 *Activity* and *Action* concepts, which do not have a notation, we suggest one.

- **Exploring approaches for UML4SPM process model executions**. In order to execute UML4SPM process models, we explore two approaches.

  The first one consists in investigating some efforts done in the area of the business process management (BPM). The idea behind this initiative is to leverage the maturity level of this domain and to take advantage of process engines and tooling supports already proposed by the BPM field. To execute UML4SPM process models using these process engines, we need to transform them into a business process execution language. We opted for WS-BPEL as a target process execution language [WSBPEL 07]. Recently, WS-BPEL has become the de facto standard for process executions and many BPEL process engines are proposed, which consolidates our choice. Thus, in this work we define a set of mapping rules from UML4SPM to WS-BPEL and transformation steps in order to map UML4SPM process models into a WS-BPEL executable code. We also discuss the advantage and limits of this approach [Bendraou 07c]

  The second approach consists in defining an *Execution Model* for UML4SPM. This *Execution Model* defines execution behavior semantics of UML4SPM concepts. Thus, for each UML4SPM metaclass having execution semantics, we define its *execution class*. *Execution classes* reproduce the execution behavior semantics of UML4SPM metaclasses at runtime. This execution semantics is expressed in terms of *operations* within the *execution classes*. Since UML4SPM extends UML2.0 *Activity* and *Action* concepts, we base our work on the *Executable UML Foundation*, a work on progress at the OMG [OMG 05c]. *Executable UML* aims at defining a compact and complete subset of UML2.0 to be known as "Executable UML", along with a full specification of the execution semantics of this subset. In this work we study this specification and we draw from it the UML4SPM *Execution Model*. We also identify the set of operations and execution classes lacking by the *Executable UML Foundation* specification [OMG 06e]. Our *Execution Model* can be reused for executing UML2.0 *Activity* diagrams since UML4SPM extends UML2.0 *Activity* and *Action* concepts.

  For the *Execution Model* we propose, we provide a Java implementation Thus, UML4SPM process models can be directly executed without any transformation or configuration phase. This implementation privileges flexibility. Some important points we have taken into account are:

  - Easy extension of the UML4SPM metamodel with a minimal impact on the *Execution Model*;
  - Strong coupling of UML4SPM process models and their execution. If process models elements are modified, their execution classes are not affected (i.e., the execution class extracts data from the process element when required. Data is not duplicated within the execution class) and there is no need to interrupt or to restart the execution;
  - Concurrency of process's activity executions;
  - Connection of the process execution with external applications (e.g., GUI, business application, etc.);

  In this work we will demonstrate how these flexibility aspects are achieved through the implementation we provide.

- **Validating the UML4SPM approach**. At this aim we evaluate UML4SPM with the set of SPMLs requirements defined by the literature (cf. first objective). In the software process community there is a well-known software process example used to evaluate the expressiveness of an SPML. This example is called the ISPW-6 software process example. In this work, we also use this process example in order to evaluate the expressiveness of UML4SPM.

  For the execution of UML4SPM process models, in this work we provide a UML4SPM *Process Execution Engine*. That latter takes as input a UML4SPM process model edited with the UML4SPM *Process Model Editor* and directly executes it. Our process engine is based on the UML4SPM *Execution Model* we defined. To validate this process engine, we test it with a complete software process example that we edited within UML4SPM *Process Model Editor*.

The research objectives and contributions we introduced in this section are presented in more details along the chapters of this thesis document. The document structure is given in the next section.

Thesis structure

This thesis is further structured as follows:

- **Chapter 2** gives the state of the art of the *Process Modeling* domain. Definitions used along this document are introduced in this chapter. It also identifies the set of SPML requirements used to compare between the different SPMLs studied in this thesis. Finally, it introduces a framework we defined for the comparison and classification of the different process technology domains.

- **Chapter 3** explores how the software process modeling community can take advantage of the MDE vision. It identifies its main principles and how they can be taken into account in defining a SPML. When designing UML4SPM, we have considered these principles.

- **Chapter 4** compares the UML-based approaches for software process modeling and highlights their limits. The evaluation of these approaches is done according to SPML requirements identified in Chapter 2 and MDE principles enumerated in Chapter 3.

- **Chapter 5** presents our software process modeling language, UML4SPM. A detailed presentation of UML4SPM metamodel's classes as well as of the subset of UML2.0 elements we reused is given. It also introduces the UML4SPM notation.

- **Chapter 6** evaluates UML4SPM according to SPML requirements and MDE principles. A particular evaluation of the expressiveness of our language is done by modeling the well-known ISPW-6 process example using UML4SPM.

- **Chapter 7** presents the UML4SPM-2-WS-BPEL approach. In this chapter we give the mapping rules and transformation steps that allow UML4SPM process models to be mapped into WS-BPEL code before to be executed. The approach is illustrated through a software process example. Finally, we discuss the limits of this approach that make us exploring alternative solutions for UML4SPM process model executions.

- **Chapter 8** defines the UML4SPM *Execution Model*, an alternative solution to the UML4SPM-2-WS-BPEL approach. In this chapter, a detailed description of the

UML4SPM *Execution Model* classes is given. We also present our Java implementation of this model and we highlight approach's positive aspects that overcome the execution approach presented in Chapter 7. The approach is evaluated through the same software process example used in Chapter 7. This process example is edited in the UML4SPM *Process Model Editor* and executed with the UML4SPM *Process Execution Engine* which both are presented in this chapter.

- **Chapter 9** concludes this document by outlining the main contributions of this thesis and by drawing some perspectives for further investigations.

Chapter 2

Process Modeling

1. Introduction

One of the most challenging tasks while exploring the *Process Modeling* domain is to become familiar with the myriad of concepts, acronyms and definitions it proposes. The aim of this chapter is to clarify some of them but most of all, to put in place some definitions that we will use along this document. The main difficulty we encountered while collecting these definitions was to pick what we believed to be the right definition among many others. As the domain is vast and brings together various sub domains and communities, the same concept may have a slightly different sense from one community to another as sometimes, it can be in complete contradiction.

Another purpose of this chapter is to present and to clarify the relationship between different process modeling domains such as *Software Process Engineering*, *Business Process Modeling* and *Workflow Management*. At this aim, we define a framework that highlights main characteristics, modeling objectives, commonalities/distinctions and scope of each domain. It also points-out the relationship of each domain with the other domains.

In the following, we start by introducing some basic concepts like "*Process*", "*Process Model*" and "*Process Modeling Language*". Then, we present principal requirements expected from a *Process Modeling Language*. These requirements will serve us in the next chapter in order to compare between different software process modeling languages. We conclude this chapter by a discussion on different process characteristics, which may vary from one domain to another.

2. Basic Concepts

The set of the so-called "basic concepts" may differ from one audience to another. In this section, we only address concepts and definitions that we believe essential for the understanding of this document. To provide these definitions we explored a multitude of sources in the literature, trying each time to exp lain the meaning and properties highlighted by the definition. A comparison of various definitions for the same concept is also given. For readers who want more definitions and taxonomy of the *Process Modeling* domain, they can be found in [Dowson 91] [Humphrey 92] [Conradi 92a] and [Lonchamp 93].

2.1. Process

The concept of *Process* is not new. It exists since the first manufacture appeared, a long time ago before even the first computer was designed. Indeed, laying out inter-related activities in a sequence and creating a flow of work has been part of organization designs for more than two centuries. Nevertheless, the use of the term *Process* varies from an organization to another and never stopped to evolve. It moved from a guide that helps in organizing the realisation of product into a means of reasoning, evaluating and enhancing the quality of the delivered products.

The International Organisation for Standardisation (ISO) provides this generic definition: "*a process uses resources to transform inputs into outputs. In every case, inputs are turned into outputs because some kind of work, activity, or function is carried out*"[ISO 98]. The term *Process* is used in various domains. A process can be administrative, industrial, agricultural, governmental, chemical, mechanical, electrical, and so on. In the following, we only consider the use of the term *Process* in the area of computer and information science and more especially, in most mature communities that deal with the process technology:

2.2.1. Process definition by the Workflow Management community

The Workflow community primary goal is the automation of business procedures or "workflows" within organisations during which documents, information or tasks are passed from one participant to another in a way that is governed by rules or procedures [WFMC 06]. The WFMC (Workflow Management Coalition), the standardization organization leading this community defines a process as "*a formalized view of a business process, represented as a coordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal* " [WFMC 99]. As we can notice, this definition may seem ambiguous. It defines a *process*, as the representation of a *business process*. In fact, the same specification defines further a *Business Process* as "*a set of one or more linked procedures or activities, which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships*". A business process in the context of Workflow is typically associated with operational objectives and business relationships, for example insurance claims process, shipping order process and so on. It may consist of automated activities, capable of workflow management, and/or manual activities, which lie outside the scope of workflow management. Looking at the WFMC definitions and at the organization purpose, we can distinguish two properties that characterize this domain. The first one is, the emphasis on the automation of business procedures and the automatic routing of documents (artifacts) to actors having predefined roles [Totland 95]. The second one is the organizational context of this kind of processes which most often is application/department specific.

The OMG's (Open Management Group) WMFS1.2 standard (Workflow Management Facility Specification) defines a process in the context of the workflow management as "*a set of discrete activity steps, with associated computer and/or human operations and rules governing the progression of the process through the various activity steps*". This definition emphasizes the important role of the governing rules, commonly called *business rules* and which represent the mechanism by which the process automation is ensured.

2.2.2. Process definition by the Business Process Management (BPM) community

In this community, which also covers *Business Process Reengineering* (BPR), people use to employ the term *Business Process* instead of *Process*. The Google search engine returns more than seven hundred millions of entries in response to the request: *"Business Process"*, which makes it difficult to converge to a unique definition. At its most generic, a *Business Process is a collection of activities that are required to achieve a business goal and it is represented with an activity flow that specifies the* *orchestration needed to complete the goal* [Bastida 05]. This definition does not really differ from the one given by the workflow community. We can notice the new term: *orchestration*, which in the BPM domain means all ordering and timing constraints that should be taken into account for process execution. In addition, the term *business goal*, which most often is intended to express the organization's "main" or "essential" activity, i.e., its core business.

Davenport [Davenport 93] defines a *Business Process* as "*a structured, measured set of activities designed to produce a specific output for a particular customer or market. It implies a strong emphasis on how work is done within an organization, in contrast to a product focus's emphasis on what. A process is thus a specific ordering of work activities across time and space, with a beginning and an end, and clearly defined inputs and outputs: a structure for process actions. Processes are the structure by which an organization does what is necessary to produce value for its customers*."

In our view, this is one of the most representative definitions of what could be a *Business Process*. It highlights many aspects of this family of processes. Aside the common the definition: "*a structured set of activities, etc.*", the first important aspect is that the business process focus is on the business logic of the process (i.e., how work is done), instead of taking a product perspective (i.e., what is done). This later perspective is more specific for instance to the Information System Development community. The second aspect is the notions of time and space, which in this kind of processes may vary from few seconds to years and may encompass as well the smaller enterprise unit as big corporations. The last aspect but not least, is the customer, which more often represents the recipient of the process' outcome.

If we consider the process definition of Johansson et. al. [Johansson 93], they define a process as "*a set of linked activities that take an input and transform it to create an output. Ideally, the transformation that occurs in the process should add value to the input and create an output that is more useful and effective to the recipient either upstream or downstream.*" In this definition, the focus is on the constitution of links between activities and the transformation perspective that takes place within the process upon artifacts (inputs) it handles.

Finally, the BPMI (Business Process Management Initiative), which is one of the most influent organizations in the BPM domain and which recently merged with the OMG (Open Management Group), provides through its BPMN standard -finalization underway- this process definition: "*a Process is an activity performed within a company or organization. Processes may be defined at any level from enterprise-wide processes to processes performed by a single person. Low-level processes may be grouped together to achieve a common business goal*" [OMG 06a]. Then, it defines a *Business Process* more generically as "*a set of activities that are performed within an organization or across organizations. Thus a Business Process may contain more than one separate Process. Each Process may have its own Sub-Processes*". Here, main characteristics that are highlighted in this definition are the hierarchical aspect of processes and their scope, which can be cross-organizations.

2.2.3. Process definition by the Software Engineering (SE) community

In this domain, process definitions do not thrive as in the previous one. In the software engineering community, we usually refer to processes as *Software Processes (SP)* or *Software Engineering Processes (SEP)*. Humphrey's software engineering process definition is probably the most cited one in the literature [Humphrey 89a].

Humphrey defines a software engineering process "*as the total set of software engineering activities needed to transform a user's requirements into software*". The term *Software* refers to a program and all of the associated information and materials needed to support its installation, operation, repair and enhancement. An important aspect pointed out by this definition is that functionalities and quality of the delivered software are partially based on the good understanding or not of user's requirements. The definition given by Sommerville is very close to Humphrey's one. He defines a SP as "*the set of activities and associated results that produce a software product*" [Sommerville 07].

Software Process is not to confuse with *Software Life cycle* which represents the period of time that begins when a software product is conceived and ends when the software is no longer available for use. The life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. These phases may overlap or be performed iteratively, depending on the software development approach used [IEEE 90]. However, the definition of *Software Process* complements the concept of *Software Life-Cycle* in the sense that, a software lifecycle defines the skeleton and philosophy according to which the software process has to be carried out, while a software process prescribes a precise course of actions, an organization, tools and operating procedures, development policies and constraints. Though, adopting a specific lifecycle is not enough to practically guide and control a software project.

In [Fuggetta 00], the author defines a SP as "*a coherent set of policies, organizational structures, technologies, procedures and artifacts that are needed to conceive, develop, deploy, and maintain a software product*". In addition to software engineer skills, this definition underlines principal constituents of a software process and which may affect the quality of the delivered product. By *organisation structures*, is meant, all the skills and means necessary to manage teams, to control time constraints and to coordinate process activities. During the software development, *tools* and *infrastructures* are needed. The choice of the proper technology to use can be a determinant criterion for the success or the failure of the software. Finally, the term *procedure* means all the material, guidelines and the methodological support needed during the software development.

On the standardization organism sides, the OMG, through its standard for software process modeling SPEM1.1 (Software Process Engineering Metamodel), roughly defines a software engineering process, as a complete description of its constituents in terms of *Process Performers*, *Process Roles*, *Work Definitions*, *Work Products*, and associated *Guidance* [OMG 05a]. Examples of *Work definition* can be an *Activity*, a *Phase* or *Iteration*. When writing up this document, the OMG was just about to finalize the revision of SPEM1.1, namely SPEM2.0 [OMG 04]. SPEM2.0 skipped the concept of *Process* and uses *Activity* instead. An *Activity* represents a general unit of work assignable to specific performers represented by *Roles*. An *Activity* can rely on inputs and produce outputs represented by *Work Products*. The SPEM standard will be addressed in more details in the following chapters.

Finally, ISO is currently working on a standard named SEMDM (Software Engineering Metamodel for Development Methodologies), which establishes a formal framework for the definition and extension of development methodologies for information-based domains (IBD), such as software, business or systems, including three major aspects: the process to follow, the products to use and generate, and the people and tools involved. It defines a *Process* as "*a large-grained work unit that operates within a given area of expertise*" [ISO 06].

2.2.4. Process definition by the Information System (IS) community

In the context of Information System Development, a *Process* is performed to produce a product. Products represent what shall be constructed, e.g. class diagrams, state charts, and so on. Processes (techniques) are the procedures which describe in what order the construction of the products shall be performed, e.g. "at first, identify classes and objects" to construct a class diagram, "identify states", and so on. The specificity of this domain is that the process's focus is more on the product to be delivered rather then on the techniques and role definitions employed for its production.

In [Rolland 93] the term process is defined as "*a related set of activities conducted to the specific purpose of product definition*". Both together, the set of products and their corresponding processes/techniques form a *Method* [Rolland 98].

While looking at all these definitions from the aforementioned communities, we can notice that roughly, almost of them has the same vision of the term *Process* (i.e., as the set of activities required to transform inputs into outputs). What may differ from one community into another are 1) the objectives the process tends to attain; 2) the means used by the process and 3) the result of applying the process. In fact, these distinctions are what characterize each of these communities. In Section 4 of this chapter, we give a more exhaustive discussion about the commonalities and distinctions between these communities and we attempt to clarify the relationship between each them.

Since the topic of this thesis relates to the *Software Process Modeling* domain, we need to agree on a unique definition of what could be a *Software Process*. It will be used along the document. This definition is Lonchamp's one and is, in our view, the one that covers all the aspects and components of software processes. Lonchamp defines a *Software Process* as "*the set of partially ordered process steps, with sets of related artifacts, human and computerized resources, organizational structures and constraints, intended to produce and maintain the requested software deliverables*" [Lonchamp 93]. Processes of the same nature, whatever the domain, are classified together into a *Process Model,* which is the topic of the next section.

2.2. Process Models

Since the earliest projects developing large software systems, one main concern of organizations was to provide a conceptual scheme for rationally managing the complexity of software development activities [Scacchi 01]. Indeed, when several people work cooperatively on a common project, they need some way to coordinate their work. For relatively small or simple tasks, this can often be done informally, but with larger numbers of people or more sophisticated activities, more formal arrangements are needed. Furthermore, within a company or an application domain, processes of different projects tend to follow common patterns. Hence, software engineers had rapidly felt the need to capture these commonalities in process representation which describes these common features and fosters the cultural homogeneity of the community.

Hosier [Hosier 61], Royce [Royce 70] and Boehm [Boehm 76] [Boehm 87] were among the pioneers to propose models of the Software Development Life Cycle

(SDLC). These models depict how software development activities are partitioned and organized in interconnected phases and iterations. Phase specifications, their ordering and the way they might be linked are proper to the life cycle model. The "Waterfall", "Spiral" and the "Incremental Model" are well known examples of SDLC.

However, the software process modeling community was unsatisfied with using these life-cycle descriptions as process models. The granularity of process steps included in SDLC models is too coarse-grained and fails to describe elementary process building blocks [Curtis 92]. Most life-cycle descriptions represent an extremely abstract model of software development and do not provide clear guidance on how to integrate the many process steps that project staffs perform. Rapidly, the need to describe in more details processes that software companies are actually performing during software development or maintenance emerged. The idea was to decompose these SDLC descriptions into sufficient detail so that they can provide more explicit guidance for executing a software development project. This is how the notion of *Process Models (PM)* appeared.

2.2.1. Process Model: Definition

Based on Curtis's definition, "*a Process Model (PM) is an abstract description of an actual or proposed process that represents selected process elements that are considered important to the purpose of the model and can be enacted by a human or machine*"[Curtis 92]. Thus, a process model is a description of a process at the type level. Since the process model is at the type level, a process is an instantiation of it. The same process model is used repeatedly for the development of many applications and thus, has many instantiations. One possible use of a process model is to prescribe "how things must/should/could be done" in contrast to the process itself which is really what happens. A process model is more or less a rough anticipation of what the process will look like. What the process shall be will be determined during actual system development [Rolland 98].

In the context of software development, a *Software Process Model (SPM)* represents a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution [Scacchi 01]. They are representative of a family of software processes expressed in a suitable formalism (i.e., diagrams and notation, code, etc.). Such models can be used to develop more precise and formalized descriptions of software life cycle activities. More often, their power depends from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Among the forms of information that people ordinarily want to extract from a process model are what is going to be done, who is going to do it, when and where it will be done, how and why it will be done, and who is dependent on what its being done [Curtis 92]. This is why the availability of a precise process model is paramount, since it provides a non-ambiguous basis for communication about the process.

2.2.2. Basic uses of Process Models

Goals that motivated the introduction of process models are manifold: they span from informal support to direct assistance in process assessment, enactment and improvement. Research on software process modeling and process models supports a wide range of objectives [Kellner 89] [Riddle 89] [Curtis 92]. Herein, we present basic uses for process models and their imperatives with the assumption of the availability of their computer-supported representation:

- *Facilitate human understanding and communication*: requires that a group be able to share a common representational format
- *Facilitate process reuse*: process development activities are time consuming. In the software process community, the goal is the repeatability of the process with optimal human resource consumption, focusing on how the job should be done. Process reuse requires the identification of the good abstraction using the right process elements (i.e., roles, products, guidance, etc.).
- *Support process improvement*: requires a basis for defining and analyzing processes and that processes be precise, easily understood and expandable.
- *Support process management*: requires a clear understanding of plans against which actual process behaviors can be compared and the ability to precisely characterize process status against them.
- *Automate process guidance*: requires automated tools for manipulating process descriptions, indirect support, like information on the current state of the process, the meaning of decision points, etc.
- *Automate execution support*: requires a computational basis for controlling process behavior within an automated environment, the identification of automatic invocations of non-interactive tools, etc.

According to Humphrey [Humphrey 89b], in order to ensure these goals, process models must:

- Represent the way the process is actually (or is to be) performed;
- Provide flexible and easily understandable, yet powerful framework for representing and enhancing the process;
- Allow to be refined to whatever level of detail is needed.

2.2.3. Granularity of Process Models

To be most effective in supporting the objectives of process modeling presented above, process models must go beyond representation. The understanding of process participants about the contents and sequencing of process steps depends strongly on the degree of details provided in the process model. The granularity issue involves the size of the process elements (i.e., steps, roles, artifacts, etc.) represented in the model. The granularity of a process step needed to ensure process precision will depend on the purpose of the model and attributes of the person that must execute the process [Curtis 92]. Recently, the pressure for greater granularity (i.e., more details) in process models is driven by the need to ensure process precision, the degree to which a defined process specifies all the process steps needed to produce accurate results [Humphrey 92]. Another pressure comes from the increasing demand for process automation, which requires precise process models at relatively deep levels of detail.

In the absence of precise process models, process participants are expected to have the appropriate knowledge and reasoning to translate these abstract process models into effective actions. This may be feasible for small and repeatable processes. However, for larger projects, this situation becomes rapidly unmanageable and often leads to a misunderstanding and to diverse interpretations of the same process model. Another consequence of imprecise process models is that this penalizes the technology transfer of companies. Indeed, process models encodes a part of the company expertise which otherwise is only in employee heads. Having detailed process models allows companies to:

- Master their assets;
- Transfer more easily the know-how to newcomers ;
- Increase repeatability;
- Allow process analysis and improvement.

For describing process models, we need some formalism, which may be graphical, code-like, or both. These formalisms in the context of process modeling are called *Process Modeling Languages (PML)*. We introduce them further in this chapter.

2.2.4. Process Model Views

Depending on how large and complex the development process is, many aspects of the process have to be modelled for the understanding of process participants. Obviously, one view cannot cover all the details of process contents. Thus, multiple views on the process are provided, each view focusing on a specific interest or aspect. Typical examples of views are:

- **The activity view**, also called the workflow view, which focuses on the types, structure and properties of the activities in the process and their relations, sequencing. This view may be used for instance, by the project manager for scheduling purposes and monitoring;
- **The product view**, which describes the types, structure and properties of the software items of a process. This view can used to see the transformation perspective of process artifacts form one state to another. It may be of interest also to the user, e.g. to understand which kind of documentation will be delivered with the software system;
- **The resource view**, that describes the resources either needed from or supplied to the process, which is relevant from a managerial perspective;
- **The role view**, which describes a particular set of resources, namely skills that performers supply and the responsibilities they accept. This is relevant to the organisation and the quality assurance personnel, besides other performers.

Notice, that one view can refer to some concepts defined in other views. One consideration then to take into account, is to ensure the global consistency of the different views. It is meant by consistency here, that a process element should have the same name and properties from one view to another.

2.3. Process Modeling Languages

A *Process Modeling Language (PML)* is a language used to express process models [Zamli 01]. Consequently, the understandability, precision and usability of a process model will mainly depend on the PML used to describe it. Requirements and design goals for a PML are mainly driven by the domain and complexity of processes being modeled. A good PML design assumes that we understand the domain, so that we can make sensible decisions on which process elements should be covered where and how. Examples of some considerations that have to be taken into account while designing a PML are:

- What has to be defined within a process model?
- What are the constituents of a software process and how are they interrelated?
- Is the process model intended to be used for description purposes? For its execution? For reasoning, analysis, improvement?

A PML can be ***formal***, ***semi-formal*** or ***informal***. A formal PML is one which is provided with a formal syntax and semantics. Semi-formal languages usually have a graphical notation with formal syntax, but not formal semantics i.e. not being executable. Natural languages, such as English, may be used as informal PML [Conradi 99].

In Section 3, we address in more details PMLs, their characteristics and requirements. A state of the art of some *Software Process Modeling Languages (SPML)* is presented in the next Chapter.

2.4. Process Metamodel

Metamodels are used to describe and analyse the relation between concepts. A model is an abstraction of phenomena in the real word and a metamodel is yet another abstraction highlighting properties of the model itself [Van Gigch 91]. A *Process Metamodel* is a conceptual framework that gives a precise definition of the constructs and rules needed for expressing and composing *Process Models* [Lonchamp 93]. While a process model represents a description of a family of processes to be (or actually) performed, the process metamodel defines the set of concepts and their relationships to be used in process models. Then, a process is an instantiation of a process model which is (should be) in conformity with its metamodel. Process metamodels represent also a powerful mean for comparing and reasoning about different PMLs. The grammar defined by each metamodel will depend on the context and modeling objectives. This topic is addressed in more detail in the next chapter in the context of metamodel-based software process modeling languages.

2.5. Process-Centered Software Engineering Environments (PSEEs)

Process-centered Software Engineering Environments (PSEEs) or (PCSEEs) are meant to support the development process. A PSEE provides a variety of services, such as assistance for software developers, automation of routine tasks, invocation and control of software development tools, and enforcement of mandatory rules and practices [Ambriola 97]. Information needed to provide such services are defined in *Process Models* which represent the main input parameter of PSEEs. Process models specify how people should interact and work, how and when automated tools are activated or invoked. A PSEE takes then as input a *Process Model* and "behaves" according to what it is defined within that model. Obviously, the PSEE is characterized by the PML that defines its input process models.

PSEEs may have different user support goals. Herein, principal one:

- *Passive role*: The user guides the process and the PSEE operates in response to user requests;
- *Active guidance*: The PSEE guides the process and prompts the users as necessary, reminding them that they should perform certain activities. The users are still free to decide if they will perform the suggested actions or not;

- *Enforcement*: The PSEE forces the users to act as specified by the process model;
- *Automation*: The PSEE executes the activities without user intervention.

Often, the same PSEE adopts more than a single form of user support [Cugola 98].

3. Process Modeling Language Requirements

In [Kellner 91a], authors concluded that the suitability of a process modeling language will depend on goals and objectives of the resulting model. A conclusion which remains valid nowadays. Research on software process modeling identified many of these objectives [Riddle 89] [Curtis 92] [Jaccheri 99]. They vary from facilitating human understanding to providing automated execution support.

Roughly, PM community viewpoints are divided into two families. Those that consider a PML as any Modeling Language (ML) and those that view a PML as any Programming Language (PL). That latter perspective was essentially supported by Osterweil's work in his well-known paper "*Software Processes are Software Too*" [Osterweil 87]. Principal PML requirements promoted by this community are *Executability* and *Formality*. While authors like Conradi [Conradi 95] and [Kellner 89], which support the first perspective (i.e., a PML as any ML), establish *Analyzability*, *Understandability* and *Modularity* as general PML requirements. Those requirements are confirmed by Armenise et at. in [Armenise 93], which add the *Generecity* requirement.

In the following, we give a brief description of principal PML requirements:

- **Formality**: The syntax and semantics of a PML may be defined formally, i.e. precisely, or informally, i.e. intuitively. Formal PMLs support, for example, reasoning about developed models, analyzing of the precisely defined properties of a model, or transforming models in a consistent manner;
- **Understandability**: It dependents on the possible process model's users. A PML should be user-oriented and easy to comprehend. Users with a computer science background will find easier to understand a model written in a PML that resembles a programming language. Those with other backgrounds may prefer graphic representations based on familiar metaphors;
- **Expressiveness**: Indicates whether all aspects of a process model may be directly modelled by language features of the PML or have, for example, to be expressed by means of additional comments. This requirement is addressed in the context of Software PMLs in the next section (cf. section 3.1);
- **Abstraction**: An abstraction mechanism allows one to focus on the important aspects of a system while irrelevant details remain hidden. Abstraction is important in process modeling, because it helps in mastering the complexity of the process by allowing the designer and user of the process to concentrate in what is important in each phase of the software development;
- **Modularization**: The PML may offer modelling-in-the-large concepts, such as modularization, to structure a process model into sub-models connected by certain relationships. With the expansion of new ways of working such as outsourcing, contracting-out, company fusions, this requirement is taking more and more importance;

- **Generecity**: provides the way of describing a general solution for a set of related problems, by parameterizing it with respect to its possible instantiations [Armenise 93]. This goes through the definition of more general, abstract sub-models which are customized within a concrete process model. In addition, a PML may offer the possibility of distinguishing between generic and specific process models;
- **Executability**: The PML may support the definition of operational models. These are executable. This implies that the PML should provide concepts and structures with an operational semantics, a key for process model executions;
- **Analyzability**: A PML, as most modeling languages, should be sufficiently formal to allow precise modeling, analysis and simulation. Reasoning about process models is a key for process improvement;
- **Reflection:** The PML may directly support the evolution of process models. In this case there are parameterization, dynamic binding, persistency and versioning issues to be addressed;
- **Multiple conceptual perspectives/views**: The PML may support the definition of views of certain perspectives of a process model. This implies mechanisms to integrate different views of a process model into an overall process model.

PMLs can be evaluated according to these requirements. However, an important observation is that some desired requirements may be in conflict so it is not possible to address all of them within one PML [Ambriola 94] [Perry 89]. Thus, fundamentally different PMLs and notations may be needed to cover such diversity in scope [Conradi 95].

3.1. Constituents of Software Process Models

As we emphasized in Section 2.3 (Process Modeling Languages), the expressiveness of a PML depends on the set of concepts that forms its vocabulary. A PML, depending on the domain, should support the description of several concepts that characterize the development process. In the context of Software Process Modeling (SPM), early classifications of the constituents of software process models have been proposed in the literature [Dowson 91] [Conradi 92a] [Feiler 93] [Lonchamp 93] [Fuggetta 00]. We give here an essential summary of each element:

- **Activity**: A concurrent process step, operating on artifacts and coupled to a human agent or a production tool. It can be at different abstraction levels i.e., activities can be decomposed. They can be at almost any level of granularity. Artifacts constitute the operands (inputs/outputs) of activities [Conradi 99]. Synonyms of Activity are Task, Step, Work definition, etc. When we say synonym, we mean from one PML to another. Of course, we can have all or parts of these synonyms as concepts of the same PML. As an example, SPEM1.1 which defines Activity, Step and WorkDefinition. Each concept with its own semantics [OMG 05a]. SPEM1.1 will be addressed in detail in the next chapter;
- **Artifact**: A product created or modified during a process either as a required result or to facilitate the process. They are the input and output of activities. An artifact can be simple or composite and may have relationships i.e.,

dependencies with other artifacts. Synonyms of Artifact are Product in [Conradi 92a], WorkProduct in [OMG 05a] and [ISO 06], Resource in [Cass 00];

- **Role**: Defines rights (i.e., permissions) obligations and responsibilities of the human agent involved in the software activity. A Role is a static concept while the binding between a role and an agent can be dynamic [Conradi 95]. A role can be played by several agents and inversely, an agent can play several roles;
- **Human**: Human are process agents who may be organized in teams. They have skills and authority and can fulfil a set of roles. They are in charge of executing certain activities that compose the process. Synonyms of Human are Agent, Performer, etc;
- **Tool**: Relates to any tool used by the software process, may be batch (i.e. compilers, links, parsers…) or interactive (i.e. textual editors, graphical CASE tools…).

Of course this set of concepts is not restricted to those defined here and may differ depending on the modeling domain. Then, as we've noticed in the previous section, the *expressiveness* requirement is tightly related to the ability of the PML to express constituents of real software development processes.

4. Classification and Comparison Of Process Technology Domains

Since organizations recognized the benefit of capturing their processes in formal representations, their interest to the *Process Modeling* discipline never stopped to grow. The promises of cost effective and better quality products and services stimulated many research activities and projects. The process technology was first developed within the manufacturing domain and rapidly succeeded to draw the attention of many others. The computer science domain counts many research areas that deal with process modeling. Most mature ones are:

- Software Process Engineering (SPE);
- Workflow Management (WfM);
- Business Process Management (BPM);
- Information Systems Engineering (ISE);
- Enterprise Application Integration (EAI).

However, whether these communities have distinct goals, they all share the fact that they handle and manage processes. This common point became rapidly a source of misunderstanding and ambiguities since each community started to give its own definitions regarding the process technology. This led, in a first time, to the proliferation of taxonomies that aim at giving definitions of basic concepts. As we saw earlier in this chapter, the term *Process* has different meaning from one community to another. In the literature, we can mention most known contributions in providing taxonomies for Software Process Engineering [Lonchamp 93] [Humphrey 89a], for Business Process Management [Davenport 93] [Scheer 99] [Giaglis 01], for Information Systems Engineering [Sol 92], and [WFMC 99] [Georgakopoulos 95] for Workflow Management.

In a second time, the process modeling community saw the appearance of a multitude of *Process Modeling Languages* within each community. PMLs are the means to capture real processes in *Process Models*. The principal aim behind modeling processes was to capture the company expertise and to ensure the repeatability of processes. Once companies had their processes described in formal representations, their objectives moved from simply representing processes to how to provide means and techniques to analyse them, to improve them and to execute them. In order to evaluate PMLs, frameworks were proposed [Curtis 92] [Paulk 95] and a set of requirements on PMLs and basic process model elements were fixed [Dowson 91] [Conradi 92a] [Jaccheri 99] [Kellner 89]. Within each community, efforts were made to compare between the different PMLs and many surveys was provided. We can quote [Conradi 99] [Armenise 93] [Zamli 01] in the context of Software PMLs, [Giaglis 01] [List 06] for a comparison of Business PMLs and [Lei 97] [Bolcer 98] [Mühlen 99] for Workflow formalism evaluations.

However, whether each community is evolving its process technology individually, few works have been done in order to find commonalities/distinctions between the different research areas. This situation led that nowadays, many process analysts and company deciders are confused about which technology to choose to achieve their goals. Besides, many people in the domain are unable to differentiate or to establish the relationship between a *Software Process (Software Process Engineering),* a *Business Process (Business Process Management)* and a *Workflow (Workflow Management)*. Are software processes a kind of workflows? What is the difference between a business process and a workflow? Are software processes business processes?

In the following, we try to answer these questions. In order to do so, we define a kind of framework that guides us while classifying and comparing these different research areas. The main goal behind this framework is to guide process modellers and deciders in their choice of the appropriate technology in regard with their process modeling objectives. Conversely, it can be used in order to identify the set of concepts and requirements that have to be respected in order to define a new PML according to the domain (i.e., SPE, BPM or WfM). The framework consists in responding to the following questions:

1. What it is the domain context and scope?
2. What are goals and objectives of modeling processes in this domain?
3. What are the domain and process characteristics?
4. What are the process modeling concepts of the domain?

In the following, we detail each of these concerns with regard to *Software Process Engineering (SPE)*, *Business Process Management (BPM)* and *Workflow Management (WfM)* domains. The choice of these domains is related with the scope of this thesis and with the fact that some domains are more mature than others. Definitions of the term *Process* within each of these domains was already addressed in Section 2.1. In [Totland 95], we can find a similar initiative. However, in the paper, authors only answered the questions (1) and (2) and did not establish any relationship between the different process modeling domains.

4.1. Software Process Engineering Domain

As introduced by Humphrey, Software Process Engineering refers to "*the total set of software engineering activities needed to transform user's requirements into software*"[Humphrey 89a]. This process may include, as appropriate activities of: requirement specifications, design, implementation, verification, installation, operational support, and documentation.

Goals and Objectives of Modeling Software Processes

The goal of software processes is to facilitate and to support the development of high-quality products more quickly and at lower cost. Modeling of software processes can have several purposes. Most important ones remain ensuring process understandability and communication between software developers. In [Armenise 93], authors add the following objectives: process planning, analysing, measuring, configuring, reusing, executing and improving. Software processes are formed of two kinds of processes. The software production process, which represents the process being actually performed by software developers and tools, and the *Meta-Process,* which consists of the activities of modeling the process, managing the process, support for its execution and improvement. Processes and meta-processes are operated by humans. One output of a process is the feedback from its operating people on the procedures and tools used. This feedback is used by the meta-process to improve the process itself by modifying the process model [Conradi 92a].

Domain and Process Characteristics

Software processes have characteristics that make them different from typical production processes. Software production is a highly creative task and therefore, it is not completely formalizable. Consequently, the execution of the activities involved in a software process cannot be done entirely by computers [Armenise 93]. According to [Ruiz 04] and [Sutton 95a] the special nature of software processes can be defined as follow:

- They are complex;
- They are exception driven, highly unpredictable since they depend too much on too many people and circumstances;
- Not all activities are supported by automated tools. Some of them may be incomplete and informal;
- They are finding-based and depend on communication, coordination and cooperation within a predefined framework;
- Their success depends on user involvement and the coordination of many roles;
- They may take a long time and are subject to changes during this time.

Of course, these features that characterize software processes have to be taken into account while designing Software PMLs.

Process Model Elements

In the literature, we can find an agreement about principal process elements that we can find in software process models [Dowson 91] [Conradi 92a] [Lonchamp 93]. Main ones are *Activities, Artifacts, Roles, Human (or agent)* and *Tools*. The description of each of these process elements was introduced earlier in Section 3.1. We can also quote notions such as *Deliverable, Constraints, Milestone, Guidance, team, Phase, Iteration, Lifecycle,* etc.

Domain context and scope

Developing software processes is generally the goal of companies that have as a main business, the production and maintenance of software. Software organizations aim at making their processes and software the most repeatable, cost effective and reliable as possible. The implication of organisations, having other business than software production, in modeling software processes would be risky and costly. Organisations doing so tend to retain their software-dependent competitive advantage by developing their own software [Thomas 95]. The critical software's components are often developed internally while non-strategic ones are bought as off-the-shelf systems. The scope of software processes in almost cases is limited to the organisation.

4.2. Business Process Management Domain

A business process is defined as group of tasks that together create a result of *Value* to the customer [Hammer 96]. We will see herein, how important the notion of *Value* is. Business Process Management (BPM) refers to the set of methods, techniques and tools to support the design, enactment, management, analysis and improvement of business processes [Van der Aalst 03b]. We can resume BPM concerns in:

- Organizing the business around processes and focusing on customer satisfaction;
- Clarifying and documenting processes;
- Monitoring process performance and compliance;
- Continuously identifying opportunities for improvement and deploying them.

Companies rely on a range of *Core* and *Support* processes, which together form the business process, to create *Value* for their customers. It is meant by value creation for customer: improved product quality, improved customer service, reduced cycle time, reduced cost to the customer. The notion of *core* and *support* processes was initially introduced by the famous Michael Porter's work in his book, "*Competitive Advantage*" [Porter 85]. Porter divided the activities within the value chain into two sets:

1. The primary activities that converted raw materials into finished products and sold and delivered them;
2. Support activities, which included technology development, human resources and firm infrastructure.

By the mid-Nineties, most authors referred to the two types of high-level processes as *core* business processes and *support* business processes.

Every business has unique characteristics embedded in its core processes that help it achieve its goals and create competitive advantage. Strategic business processes, such as new product design or high-sensitivity customer care, provide unique and durable business advantages to organizations. Those that depend on people's intelligence, experience, knowledge, judgment and creativity are the hardest for rivals to duplicate. Core processes are unique to a firm and have greater strategic importance than support processes. Support processes deal with the activities of process automation, resources and process management, analysis and improvement. Nowadays these activities are mostly known as BPA for Business Process Automation or

Analysis, BAM for Business Activities Monitoring and STP for Straight Through Processing.

If the objective is to create competitive advantage, then a company's focus should be on core processes, such as new product development or customer care and retention. Examples of questions when developing the core process are: *what can we do to permanently cut the costs of our operations? How can we boost revenue? How can we get first-mover advantage?* However, if the objective of a process is operational efficiency, then a support process may be a better choice.

In the literature, we can also find the notion of Business Process Reengineering (BPR). BPR is a management approach aiming at improvements by means of elevating efficiency and effectiveness of the processes that exist within and across organizations. It is a fundamental and radical approach by either modifying or eliminating non-value adding activities. BPM differs from BPR in that it does not aim at one-off revolutionary changes to business processes, but at their continuous evolution.

Goals and Objectives of Modeling Business Processes

Every organisation has a core business. It can range from producing and delivering of goods, producing software, to providing services as transportation or medical cares. Their principal concern remains how to augment efficiency and reliability of goods or services they provide. The notion of *Value* is then related to the degree of reliability (in case of producing goods) or satisfaction (in case of providing services) the organisation's business meets the customer expectations. In their paper "*Business Modelling Is Not Process Modelling*", authors pointed-out a very important goal of business process modeling [Gordijn 00]. They define as a main goal of having business process models, the answer to the question "*Who* is offering *What* to *Whom* and expects *What* in return". Therefore, the central notion here is the concept of *Value*, in order to explain the creation and addition of *Value* in a multi-party stakeholder network, as well as the exchange of *Value* between stakeholders. Thus, creation/addition of value for customer and for the organisation is among principal goals that can induce organisations to model and reason about their business processes. The notion of value as an important concept in business models is also pointed out in [Timmers 99] in terms of benefits and revenues.

Obviously, the support for human understanding, communication, process improvement, analysis, simulation and execution remain important parts of BPM objectives [Giaglis 01] [Ould 95].

Domain and Process Characteristics

Business processes are essentially characterized by their dependency on human intervention and by their complexity, often involving unpredictable variables that can require major changes during execution, even changing the entire course of the process. In [Jennings 96], authors emphasize the following characteristics:

- A business process crosses functional/organisational boundaries. Organisations are physically distributed. This distribution may be across one site, across a country, or even across continents. Within organisations, there is a decentralised ownership of the tasks, information and resources involved in the business process;
- Different groups within organisations are relatively autonomous—they control how their resources are consumed, by whom, at what cost, and in what time

frame. They also have their own information systems, with their own idiosyncratic representations, for managing their resources;

- There is a high degree of natural concurrency—many interrelated tasks are running at any given point of the business process;
- There is a requirement to monitor and manage the overall business process. Although the control and resources of the constituent sub-parts are decentralised, there is often a need to place constraints on the entire process (e.g. total time, total budget, etc.);
- Business processes are highly dynamic and unpredictable—it is difficult to give a complete *a priori* specification of all the activities that need to be performed and how they should be ordered. Any detailed time plans that are produced are often disrupted by unavoidable delays or unanticipated events (e.g. people are ill or tasks take longer than expected).

We can also add that:

- Business Processes are often long running;
- Business Processes need user interaction;
- Business Processes may need to migrate in the event of hardware failure or for performance;
- Business Processes are stateful;
- Business Processes have *customers, internal or external*, which may receive products or services from a business process. External customers are outside of the organisation. Core processes concentrate on satisfying their requests. Internal customers are part of the organisation. They represent other groups or departments and relate to support processes.

Process Model Elements

In addition to well-established process elements we introduced earlier (cf. Section 3.1); there are some other aspects and concepts specific to business processes. These concepts deal more with the organisation context. In [List 06], authors denote for instance the notion of *Organizational Unit* or *Entity*, of *Customer*, which can be internal or external, of *Agent*, the notion of *Software*. There are also some concepts that deal with the organisation's business context such as, *Goal* or *Objective*, *Process Owner*, *Service, Event*, *Message*, *Condition*, *Transaction*, *Sub-Process*, *Time Date* and *Business Rule*. Business rules represent the knowledge and rules in an organization that prescribe and/or restrict the way in which process activities are accomplished. Some of these rules exist in a formalized way; others exist only informally. Some rules are precisely defined, others allow for discretion of a human actor [Endl 98]. Finally as the main goal for modeling business processes is the creation of value, business process models often provide concepts for *measuring* process efficiency.

Domain context and scope

Achieving organisation business goals can make business processes span many organization's units as well as many other organizations. More often, a business process is composed of sub-processes, which are achieved by the organisation units.

4.3. Workflow Management Domain

The term *Workflow* or *Workflow Management* designates the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [WFMC 99]. The business process is defined within a *Process Definition*, which identifies the various process activities, procedural rules and associated control data used to manage the workflow during process enactment.

Workflow Management System (WfMS) are used to define, create and manage the execution of process definitions through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.

Goals and Objectives of Modeling Workflows

An important objective in workflow management is to be able to automatically route artifacts - most often documents- through a network and according to predefined rules, to actors having predefined roles [Totland 95]. This aims at saving time and money by ensuring that the right person or entity is being affected with the right tasks and documents at the right moment. Workflow management tends to answer the *"Who?"* (Business process's participants and roles), the *"What?"* (Process activities, roles have to do) and the *"When?"* (When does a role start an activity).

Domain and Process Characteristics

Since a workflow is the automation of parts of a business process i.e. automatic routing of artifacts across process activities, workflow processes are much related to the business process they support. Thus, they share few characteristics such as:

- Workflows can be long-time running. This is not the case of a majority of workflows. Often it depends of the business process they support [Schmidt 98];
- They depend on humans. Even if the process is automated, it depends on the agent (human) executing the process activity. The efficiency in terms of time and quality depends on the agent's ability and skills to do the work on time.

However, the workflow technology has some characteristics that are in contradiction with the nature of business processes, thus making their support and management limited. Herein some of them:

- Workflow systems are suitable for supporting rigidly structured, well-defined and repeatable business processes. Since processes are more often implemented in terms of proprietary code and procedures, any change in the business process, e.g., new requirement, new objective, may reveal very costly and time consuming. Besides, procedure implementations are task-specific and may require recompiling the code [Swenson 95] [Adams 03];
- Business rules in Workflow systems are hard-coded. Once the process execution is lunched, it is not possible to modify them;
- Workflow systems are tightly coupled through customized APIs with software and applications used during the process;
- Workflows are limited in modeling all business process aspects in the sense that most concepts relate to the coordination of tasks and artifacts routing;
- Not all workflow systems provide graphical representations.

Process Model Elements

As for the previous domains, the Workflow technology has some domain-specific concepts in addition to the usual ones i.e., Activity, Role, Artifact, Tool and Human. Concept names may differ but roughly, they designate the same thing. For instance, the term *Workflow Participant* or *Agent* is used instead of Human, *Application* or *Software* instead of Tool, etc. Regarding process model elements proper to workflows, we can mention *Work Item*, which designates the representation of the work to be processed by a workflow participant in the context of an activity within a process instance [WFMC 99]. Work items are normally presented to the workflow participant via a *Work List*, which maintains details of its allocated work items. We also have the notion of *Task*, which represents an automated activity, the notion of *Deadline*, *Event* or *Pre* & *Post-Conditions* and *Procedure*, *Rule*, which model process business rules.

Domain context and scope

In enterprise-wide applications, workflows may span multiple organizational units, which are often, to a large extent, autonomous. Consequently, for scalability reasons, is not unusual that an enterprise decides to partition a large workflow into a number of sub-workflows (e.g., based on organizational responsibilities) each of which can be handled by a different workflow management system. In order to allow different workflow systems to interact and to harmonise the way of exchanging data between workflow components, the WfMC defined a reference model [WFMC 95]. This reference model comes in form of five interfaces that identify workflow primary interaction modes. Interface 4 is the one dedicated to workflow system interactions.

Now we have introduced the three domains and presented their characteristics, in the following we try to clarify the relationship between each domain.

Business Process Management Vs Workflow Technology

In the process modeling community, most people still have some difficulties to distinguish the difference between the Workflow domain and the Business Process Management domain. Even if the frontier between these two domains may look very thin, it exists.

Business processes are a specific category of processes. A business process is conceptually defined as a high-level process determined by the overall goals of the enterprise [Georgakopoulos 95]. One primordial goal remains the creation/adding of value to customers. Business processes contain activities that interface with market partners (i.e., customers, suppliers, or other third parties). We refer to them as core processes.

Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal. The function of a *Workflow Management System* is to direct, co-ordinate and monitor execution of tasks arranged to form workflows [Lawrence 97]. Traditional workflow systems support the partial automated handling of small and repeatable steps within business processes. However, support is restricted to the process-oriented part of a business case (e.g., activity B starts when activity A finishes, etc.). The function-oriented part, which comprises applications realizing business functions, is usually done by other business process components [Schreyjak 98].

Business Process Management is the sum of all organizational activities centred around the definition, implementation, execution, control, supervision and improvement of processes. It is considered a more holistic view of *Business Process Reengineering* in that includes execution, measurement and control of processes, in addition the modeling and improvement or redesign activities. **BPM in general is an organizational concept. Workflow Management tends to be seen as the technical coordination of process execution. It is a component of a comprehensive BPM strategy, but does not encompass the strategic or change management activities associated with BPM. Thus, the Workflow technology is a subset of Business Process Management**.

However, it is not surprising to find in the industry and in the literature some workflow products and propositions that claim that they fully support business processes. They are commonly called Advanced Workflows. We did not explore them but in [Bolcer 98] [Swenson 95] [Becker 02] and [Georgakopoulos 95], the reader can find a set of requirements, features and tradeoffs that traditional workflow systems have to provide in order to support all phases of a business process lifecycle.

Software Processes Are Business Processes Too

Although many modern organisations are software-dependent, this does not mean that software development is necessarily a critical business process for them [Thomas 94]. Business processes are not limited to the modeling of trading activities such as "Develop market" or "Sell to customer", but include any meaningful human-work and automated activities both coordinated in order to achieve a business goal. Therefore, for companies vending software or providing software maintenance services, a software development process is considered as a critical business process. The business goal of these companies is then developing and maintaining software. All the expertise, tools and techniques used during the software development process are the means to ensure product's quality, short time to market, cost-effectiveness and customer's satisfaction. Once these objectives reached, they represent the creation of a *Value* to the customer (in terms of satisfaction) and to the organization (in terms of benefits). Thus, **Software Processes Are Business Processes Too**.

Historically, the notion of software processes as well as all the tooling and support for developing software appeared before the one of business processes. However, it is important to recognize that, in terms of modeling concepts, the business process modeling domain is richer than software process one. This is due to the fact that a business processes have to capture many organizational as well as functional aspects of the process. Organizational details concern all aspects related to the organization's business goals, customers, financial partners, human resources, organisational units, measures, etc. While the functional details deal with the coordination of process activities and participants, managing artifacts, process simulation, execution and monitoring, etc. These distinctions in modeling concepts between the two domains added to modeling goals specific to each domain (cf. Sections 4.1 & 4.2.), justify the myriad of PMLs in both domains.

More discussions on the topic can be found in [Thomas 94] [Scacchi 94] [Henderson 94] and [Gruhn 92].

Regarding the relationship between *Software Process Engineering* and *Workflow Technology*, it is the same as the one between *BPM* and *Workflow Technology*. Indeed, since software processes are special cases of business processes, **in SPE, workflows**

are used as means to automate the coordination of repeatable process steps and roles. Software process analysis, improvement and management are not supported by workflow engines.

Which Process Technology Fits the Best Your Needs?

After having introduced the different domains that deal with processes, in the following, we present our Process Technology Framework (See Table 2.1), which summarizes the characteristics, scope and constituents of each process domain. It also clarifies the relationship between each domain. As we argued earlier, this framework aims at facilitating and helping deciders and process modeller in their choices for the right process technology that will fits the best the organisation's core business and objectives. It is also a kind of a requirement book that can be taken into account if one needs to design a new PML within any of the SPE, BPM or WfM domains. We validated this framework in [Bendraou 07b].

Process Domains ➡ **Characteristics** ⬇		**Software Process Engineering (SPE)**	**Business Process Management (BPM)**	**Workflow Management (WfM)**
Definition of the term *Process*		*The set of partially ordered process steps, with sets of related artifacts, human and computerized resources, organizational structures and constraints, intended to produce and maintain the requested software deliverables* [Lonchamp 93]	*A specific ordering of work activities across time and space, with a beginning and an end, and clearly defined inputs and outputs. Processes are the structure by which an organization does what is necessary to produce value for its customers* [Davenport 93]	*A formalized view of a business process, represented as a coordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal* [WFMC 99]
Goals of Modeling Processes	***Primary Goal***	To facilitate and to support the development of high-quality software more quickly and at lower cost	Creation/addition of *Value* for the customer and for the organisation	Automatic routing of artifacts across process activities and participants
	Secondary Goals	Process planning, understanding, analysing, measuring, configuring, reusing, executing and improving	Support for human understanding, communication, process improvement, analysis, simulation and execution	Saving of time by automatic task and artifacts affectations, detection of process bottlenecks
Process Composition (Sub-Processes or Phases)		- The *Software Production Process*: represents the process being actually performed by software developers and tools - The *Meta-Process*: consists of the activities of modeling the process, managing the process, support for its execution and improvement	- The *Core Process*: the primary activities that converted raw materials into finished products and sold and delivered them - The *Support Process*: the activities of process automation, managing resource and process, analysis and improvement	There is no firm consensus in the literature on process phases in the Workflow domain. The WfMC define two phases: - *Process Definition Phase*: designates the time period when manual and/or automated (workflow) descriptions of a process are defined and/or modified electronically - *Process Execution Phase*: the time period during which the process is operational, with process instances being created and managed

Domain and Process Characteristics		- They are complex - They are exception driven - highly unpredictable since they depend too much on too many people and circumstances - Not all activities are supported by automated tools - They depend on communication, coordination and cooperation within a predefined framework - Their success depends on user involvement and the coordination of many roles - They may take a long time and are subject to changes during this time	- Business processes may cross functional/organisational boundaries - High degree of natural concurrency—many interrelated tasks are running at any given point of the business process - There is a requirement to monitor and manage the overall business process - They are complex, highly dynamic and unpredictable - They are often long running - They need user interactions - They may need to migrate in the event of hardware failure or for performance - Business Processes are stateful - Business Processes have customers (internal or external)	- Workflow systems are suitable for supporting rigidly structured, well-defined and repeatable business processes - Processes are more often implemented in terms of proprietary code and procedures - Business rules are hard-coded. Once the process execution is lunched, it is not possible to modify them - Workflow systems are tightly coupled through customized APIs with software and applications used during the process - Workflows are limited in modeling all aspects of business processes - Not all workflow systems provide graphical representations
Process Model Elements	***General Process Elements***	Activity, Artifact, Role, Tool and Agent	Activity, Artifact, Role, Tool and Agent	Activity, Artifact, Role, Tool and Agent
	Domain-specific Process Elements	Deliverable, Constraint, Milestone, Guidance, Team, Phase, Iteration, Lifecycle, etc.	Organizational Unit or Entity, Customer, Software, Goal, Process Owner, Service, Business Rule, Event, Message, Condition, Date, Transaction, Sub-Process, Time, etc.	Work Item, Work List, Task (automated activity), Instance (of process or activity), Deadline, Procedure, Rule, Application, Event, etc.

Domain's context and scope	Organisations having as main business the development and maintenance of software. In addition, organisations that tend to retain their expertise and strategic business by developing their own software. The scope of the process is the organisation.	Any organisation, what ever its business, that wants to create/add value to its customers. Business processes span many organization's units as well as many other organizations	Workflows are often Organisation/Department specific
Relationship with other domains	- Software Processes are special cases of Business Processes. - Software Processes use the Workflow technology to automate the routing of artifacts, the coordination of process activities and roles	- A Business process can be a Software Process since the BPM domain is richer than the SPE. - BPM is a superset of the Workflow technology. That latter is used to automate process's repeatable tasks and for artifacts routing	- The Workflow Technology is used by BPM and by SPE to automate process's repeatable tasks and for routing artifacts.

Table 2.1. A Framework for classifying and comparing process technology domains

5. Conclusion

In this chapter, we addressed many aspects and concepts that relate to the *Process Modeling* discipline.

First, we started by giving different and most cited definitions of the term *Process* across some mature communities that deal with process modeling. Communities we addressed are *Software Process Engineering (SPE), Business Process Management (BPM), Workflow Management (WfM)* and *Information Systems Engineering (ISE)* communities. Our intention was not limited to simply provide these definitions but to bring out and to comment specificities of each of them. The main purpose behind was to clarify the vision of each community about what they define as a *Process*. This helped us in identifying process-specific characteristics of each domain. As the topic of this thesis is about software process modeling, we retained Lonchamp's software process definition among many others as the one to use along the document. Lonchamp defines a software process as "*the set of partially ordered process steps, with sets of related artifacts, human and computerized resources, organizational structures and constraints, intended to produce and maintain the requested software deliverables*".

Then, we introduced *Process Models* and we presented the motivations for representing processes in a more formal format rather than simply using natural language. We also emphasized on the increasing need and process community pressure for having more fine-grained process models. This pressure is mainly driven by two reasons. The first one is due to the fact that since processes are more and more complex, that development teams are constantly renewed and the appearance of new way of working (outsourcing), organisation's expertises have to be captured in sufficiently precise models in order to ensure the transfer of the technology. The second reason relates to the increasing demand for process automation, which requires precise process models at relatively deep levels of detail. We will see in chapter 4 that not all languages provide fine-grained process models, and how this lack can penalize process automations.

We also discussed *Process Modeling Languages* and general requirements that have to be taken into account while designing a PML. An important observation we came to is that many of these requirements may be in conflict and often, it is not possible to satisfy all of them at once. These requirements will be used in the next chapter to compare between some software PMLs.

Finally, in front of the proliferation of process technology domains such as *BPM*, *SPE*, and *WfM*, it becomes more and more difficult to distinguish between these different domains. Each domain comes with its concepts, tools and marketing slogans such as nowadays, company managers pain to choose the right technology that should satisfy their expectations. In order to make things clearer and to ease the choice between these domains, we defined a kind of framework that classifies and compares the different process technologies (i.e., SPE, BPM and WfM) [Bendraou 07b]. This framework gives the process definition, characteristics, modeling objectives, process model constituents, process context and scope of each domain. It also clarifies the relationship between each of these domains. By means of this framework, we concluded that whatever a *Software Process*, a *Business Process* or a *Workflow*, it remains a process. What differ from one domain to another, are the process modeling objectives, the means and technology used by the process to attain these objectives and the characteristics proper to each domain. All these distinctions justify the myriad of

PMLs within the different domains. Having one massive PML in order to deal with all these domains would be inefficient since it will need to be tailored and customized to fit domain-specific characteristics.

Since in this chapter we clarified the distinctions between the different process modeling domains and since the topic of this thesis is restricted to software processes modeling, in the next chapters we only address *Software Process Modeling Languages*. Comparing Software PMLs, Business PMLs and Workflow formalisms would be incoherent since they do not address the same modeling objectives.

Chapter 3

Software Process Modeling Within the MDE Vision

1. Introduction

Since late eighties, research on software development processes and management saw the emergence of a multitude of *Software Process Modeling Languages* (SPMLs). Some of them were rules based (e.g., MARVEL) [Kaiser 90], others Petri-net based (e.g., SPADE) [Bandinelli 93] or object-oriented and programming languages based (e.g., SPELL, APPL/A) [Conradi 92b] [Sutton 95b]. They are commonly called first-generation languages. Despite the fact that these languages were executable and semantically rich, they did not gain much attention from the industry [Di Nitto 02] [Henderson 04] [Chou 02]. Their complexity, their use of low-level formalisms and the impossibility for non-programmers to use them, were among the main causes of their unsuccessfulness and of their limited spreading. The continuing proliferation of these first-generation PMLs has naturally risen in the SP research community the need of standardizing software process engineering concepts and best practices.

On the other hand, advances in software development and information processing technologies have resulted in attempts to build more complex software systems. These complex systems highlighted the inadequacies of the abstractions provided by modern high-level programming languages [France 06]. This has led to a demand from the software development community for languages, methods, and technologies that raise the abstraction level at which software systems are conceived, built, and evolved. In response to this keen interest in raising the abstraction level of programming languages, the software industry saw in models, the means to reach this goal. They represent a powerful mechanism for abstracting system specifications from the underlying technical and technological features. This has as direct effect, the increasing of understandability, analysability and model exchanges of software specifications between development teams. This initiative is commonly known as MDE: Model-Driven Engineering. MDE is promoted as an approach to software development where extensive models are created before source code is written. By considering models as first class entities, MDE aims at increasing productivity by reducing software production complexity [Bendraou 07a].

The MDE approach promotes the use of models and platform-independent modeling languages. During the last decade, only few modeling languages gained the attention of both the industry and the academia. Undeniably, UML (Unified Modeling Language) succeeded to become the *de facto* standard for modeling object-oriented applications and systems in the industry. The principal ingredients that participate in the success of UML - among others - are 1) its power of abstracting the complexity of the systems being modelled and 2) the use of an intuitive and understandable set of notations and diagrams. Nowadays, complete panoply of tools, documentations and training supports are available. A large number of engineers and software developers are already familiar with UML and with the accompanying tools.

The standardization of UML, together with the promises of the MDE approach in terms of abstraction and productivity has logically attracted the attention of the process modeling community. Therefore, the possibility of using UML as a process modeling language has been largely explored in the literature. This was not limited to the SP modeling domain, but was also addressed by the business process and workflow management communities [OMG 00a] [Bastos 02] [Manttel 05].

In this chapter, more than introducing the MDE vision, we will highlight its principles and we will focus on how the software process community can take advantage of this approach in order to gain in efficiency and to reduce the complexity of software process descriptions. As UML appears to be a main actor of the MDE and since our approach is based on UML, we will briefly present this language and the different extension possibilities it offers to be adapted for specific modeling domains. Our domain of interest in this thesis is software process modeling. The understanding of UML and its extension mechanisms is required for the comprehension of the next chapter where a survey on UML-Based software process modeling languages is addressed. The MDE principles introduced in this chapter are among the criteria on which this survey is based on.

2. Model-Driven Engineering and Software Process Modeling

Among the plenty definitions that we can find in the literature about MDE, we preferred to introduce Mellor's one. According to Mellor, "*Model-Driven Engineering is simply the notion that we can construct a model of a system that we can then transform into the real thing*" [Mellor 03]. Thus, this definition makes each of us a model-driven developer. However, one important difference between the traditional modeling practices and the MDE is that MDE's vision is not to use models only for documentation purposes (i.e., contemplative models). Models are to be used as formal input/output for computer-based tools implementing precise operations (i.e., operational models) [Bézivin 05]. The focus of software developers is then moved from programs and code into models, which become primary artifacts through the software engineering lifecycle. In its MDA (Mode Driven Architecture) guide, the OMG defines a model as the "*formal specification of the function, structure and/or behavior of an application or system*" [OMG 03]. In a broader sense, *"Modeling is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose"* [Rothenberg 89].

For an efficient use of MDE, there are some considerations to take into account, whatever the application domain i.e., systems modeling, software process modeling, etc. We address them in the below-sections.

2.1. Raising the Abstraction Level of Modeling Languages

The raise in abstraction claimed by the MDE vision can be viewed as the logical continuity that programming languages have been faced during these last five decades. Years ago, first computer programs were written in a target computer's numeric machine code that had been used with the very first computers. In the beginning of the fifties, the assembly language was introduced. Successions of "zero" and "one" in computer programs were replaced by mnemonics (a code, usually from 1 to 5 letters, that represents a machine language instruction that specifies the operation to be performed), abbreviations or words that make it easier to remember a complex instruction and make programming in assembly an easier task. An assembly language program was then translated into the target computer's machine code by a utility program called an assembler. An assembler is distinct from a compiler, in that it generally performs one-to-one translations from mnemonic statements into machine instructions while a compiler takes an entire program and translates it as a body. To a certain extent, the translation of assembly programs into a machine code can be viewed as it is most known today in the MDE domain, a *Model Transformation*. This model transformation, transforms a human-readable model written in assembly language into a machine-readable model written in a machine code. The goal behind this transformation is the operationalization and the execution of assembly programs. Nevertheless, whether the assembly code proved to be very efficient, it quickly tuned out unsuitable to deal with the increasing complexity of computer programs. This is what induced the introduction of High-Level Programming Languages. "High-Level Language" refers to the higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays and complex arithmetic or boolean expressions. In addition, they have no machine language instructions that can directly compile the language into machine code, unlike low-level languages like the assembly language. In order to execute programs written in a high-level programming language, programs may need to be first translated (or compiled) into assembly language or byte code and then to machine code by means of virtual machines.

Thus, by definition, every time a programmer writes a program in Java or C++, there is actually a succession of model transformations which are performed before that the program is executed. MDE then, can be considered as a natural continuation of this trend. Instead of requiring developers to use a programming language spelling out "*how*" a system is implemented, it allows them to use models to specifying "*what*" system functionality is required and what architecture is to be used [Atkinson 03]. This move to a higher abstraction aims at reducing complexity issues related to the independency from platforms and languages, distribution, interoperability, persistency, etc. However, one crucial challenge to be considered for leveraging this raise in abstraction is to be always able to go a level down while minimizing the loss of data and preserving code efficiency. Much more challenging, is to be able to go the other way up in order to reflect (trace-up) changes applied at the low-level. Figure 3.1, summarizes quite well the relationship between the language's abstraction level and the degree of abstraction needed for modeling a subject matter under study. The main idea is that one starts modeling an abstract problem (a bank application is given as an example) using an abstract language (e.g., the Unified Modeling Language). Using the abstract language will considerably ease the comprehension the of the problem since developers abstract away object allocation issues, exceptions handling, etc. The resulting system model is then to be refined, completed and transformed progressively until it ends as a solution of the problem in a low-level concrete programming language such as Java or C++.

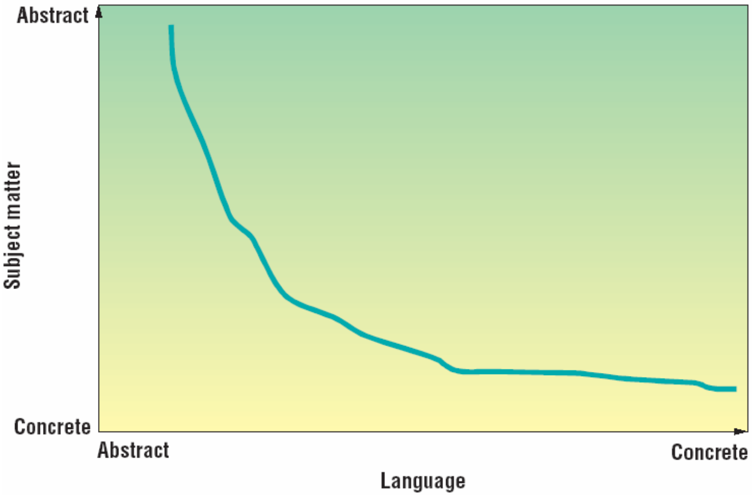


Figure 3.1. Relationship between a language's abstraction level and the degree of abstraction of the subject matter under study. From [Mellor 03]

In this work, we aim to apply MDE to the Software Process Modeling domain. As we introduced in the previous section, first-generation SPMLs, whatever the formalism they used e.g., Petri Nets, inference rules or programming languages such as Ada were considered as relatively low-level process modeling languages. Indeed, aside to be executed, the principal goal behind modeling software processes is first to enable people reasoning, understanding and communicating about tasks to accomplish during the development process. A software process described using the Ada language for instance may be understandable in case of a simple process but it quickly turns out very complicated for complex software processes. Software process executions have to deal with many aspects such as artifacts management, events, exceptions handling, control flows, etc, which may not be of primary importance while describing the process to software developers. MDE can help process modeller to raise the abstraction level of SPMLs they use to model software processes in order to concentrate on the process domain aspects while keeping a tight relationship between process models and the way they can be executed.

2.2. Executability of Models

Since the main motivation of the model-driven development is to improve productivity, executability of models appears to be a pivotal requirement. Like a programmer who writes a program and then tests it straightforward by a simple click, a modeler should be able to test its models as easier as that. In [Harel 00], the author compares models that cannot be executed to cars without engines. One important advantage of executable models is that they can provide an early direct experience with the system being designed [Selic 03].

Making software process models executable would allow, at earlier stages, to check many aspects of the process. First, the process modeler can check the right sequencing of process's activities and properties such as the process termination (e.g., does the process end one day?) or activity starting (e.g., does this activity start one day?). Second, the availabilities of resources and roles involved in the process are checked beforehand (e.g., three designers, one tester, two analysts, etc.). Third, to be able to reason about the process and to identify some process deadlocks or time-consuming activities.

2.3. Metamodelling

To be productive, models should be usable by machines [AS CNRS 04]. Thus, the language used to define these models has to be defined precisely. Metamodels are used at this aim. A "*metamodel is a model that defines the language for expressing a model*" [OMG 07a]. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language [Seidewitz 03]. The relation between a model and its metamodel is also related to the relation between a program and the programming language in which it is written, defined by its grammar, or between an XML document and the defining XML schema or DTD [Bézivin 05]. This relation is a "*Conforms To*" relation and a model conforms (or is in conformity) to (with) its metamodel if the elements defined in the model as well as relationships between these elements are defined in the metamodel.

Analogous to modeling, metamodelling has many advantages. Metamodels help abstracting low level integration and interoperability details and facilitate partitioning problems into orthogonal sub-problems. Hence, metamodels can serve as an abstraction filter in a particular modeling activity [Bézivin 01], as devices for method engineering [Brinkkemper 01], language modeling, and conceptual definition of repositories and Case tools [Talvanen 02]. Because a metamodel is a model itself, we express it in some modeling language. One of the most known metamodelling languages is the OMG's MOF (Meta Object Facility) [OMG 06c].

2.4. Standardization

Standardization provides a significant impetus for further progress of the MDE trend. It codifies best practices, enables and encourages reuse, and facilitates interworking between complementary tools. It also encourages specialization, which leads to more sophisticated and more potent tools [Selic 03].

In reaction to the proliferation of languages and propositions claiming their support for MDE, by early 2000, the OMG introduces its particular variant of MDE under the MDA^{TM} acronym for Model Driven Architecture [OMG 03]. MDA promotes the OMG shift from "*every thing is object*" to "*every thing is a model*" and may be defined as the realization of MDE principles around a set of OMG standards like MOF [OMG 06c], XMI [OMG 05b], OCL [OMG 06b], UML [OMG 07a, OMG 07b], SPEM [OMG 05a], etc. MDA encourages the use of MOF as a foundation for building new languages, the MOF/QVT standard for transforming models from one source language into a target language, XMI as an interchange formalism and finally UML as the reference language for defining either Platform-Independent Models (PIM) or Platform-Specific Models (PSM) of systems and applications. The MDA is not the only initiative for tackling the MDE vision and many others approaches tend (or already anticipated) to participate in the "every think is model" shift. Among them, the Microsoft Software Factories (SF) proposition [Greenfield 03] and the Model-Integrated Computing (MIC), which is probably among the pioneer in this trend [Sztipanovits 95].

In the area of software process modeling very few standards exist comparing to the Workflow and Business Process Management domain. We count only two propositions, the ISO SEMDM (Software Engineering Metamodel for Development Methodologies) standard, which is currently under standardization [ISO 06] and the OMG's SPEM (Software Process Engineering Metamodel) standard. SPEM is addressed in more detail in the next chapter and the ISO proposition is discussed briefly.

In this section we presented main considerations to take into account of a better leveraging of the MDE vision. The aspects we introduced here (*Abstraction*, *Executability* of *Models*, *Standardization*, *Metamodelling*) combined with SPML requirements that we will introduce in the Chapter 4 constitute our basis for combining MDE advantages and SPM knowledge for an efficient and a better support of the software process modeling discipline. In the next section we introduce UML, a main actor of the OMG's MDA. Since our proposition is based on UML, we thought that it is essential to present its principal facets.

3. The Unified Modeling Language

Since its introduction by the OMG in 1997, the Unified Modeling Language (UML) has become one of the most widely used standards for specifying and documenting information systems. In its new version, i.e., UML2.0, the standard's designers emphasized on the better support for the notion of UML as *a family of languages*. This notion is mainly ensured through the use of profiles and semantic variation points that mark the part of UML intentionally left without semantics to accommodate user-defined ones. An effort was also done in order to improve expressiveness, including improved modeling of business processes and the integration of the Action Semantics that developers can use to define the model's runtime semantics and provide the semantic precision required to analyze models and translate them into implementations [France 06]. The UML2.0 Actions are presented in more details while presenting our approach in Chapter 5.

The standard comes in four parts:

- Infrastructure which defines base concepts that provide the foundation for UML modeling constructs [OMG 07a]
- Superstructure introduces the concepts that developers use to build UML models specifying their systems and applications [OMG 07b]
- Object Constraint Language which defines the language for expressing invariant conditions that must hold for the system being modeled or queries over objects described in a model and operation. OCL can also be used to specify operations / actions that, when executed, do alter the state of the system [OMG 06b].
- Diagram Interchange which enables a smooth and seamless exchange of documents compliant to the UML standard (referred to as UML models) between different software tools [OMG 06d].

Even if UML tend to cover many developers modelling preoccupations, there are situations, however, in which a language designed to be of such broad scope may not be appropriate for modelling applications of some specific domains such as for instance, process modeling, architecture modeling, etc. This is what justifies the many propositions of domain-specific languages (DSLs) and approaches such as the Microsoft's Software Factories. In some domains, the UML syntax or semantics cannot express specific concepts of particular systems, or where we want to restrict or customize some of the UML elements which are usually too abundant and too general [Fuentes 04].

In order to deal with the modeling of domain-specific concepts, the OMG defines two possible approaches for defining DSLs: the *heavyweight extension* mechanism and the *lightweight extension* mechanism. Since, in this thesis we propose to reuse UML for software process modeling, in the following we present each of them with their respective advantages and drawbacks.

3.1. Heavyweight Extension

The heavyweight extension, also referred to as first-class extension consists in defining a new language (i.e., an alternative to UML), using the MOF (Meta Object Facility) standard. The syntax and semantics of the new language's elements - that do not match with semantics of UML elements - , their properties, operations and relations between these elements are defined to fit the specific characteristics of the targeted domain.

The MOF provides also an extension mechanism that allows defining a new metamodel from an existing metamodel. Commonly, the new metamodel is defined by importing (i.e., using the <<import>> relation) classes from packages of an existing metamodel. These classes are then extended by new domain-specific classes through the specialization /Generalization mechanism [Desfray 99]. An example of this kind of extension is the SPEM1.1 standard, which is built as a MOF stand-alone metamodel that some of its classes extend UML1.4 metamodel classes (see figure 3.2). The concepts proper to software process modelling are regrouped in the *SPEM_Extensions* package which extends a subset of UML1.4 metamodel contained in the *SPEM_Foundation* package (Figure's 3.2. left part). Examples of classes from the software process domain extending UML1.4 classes are given in the right part of the figure.

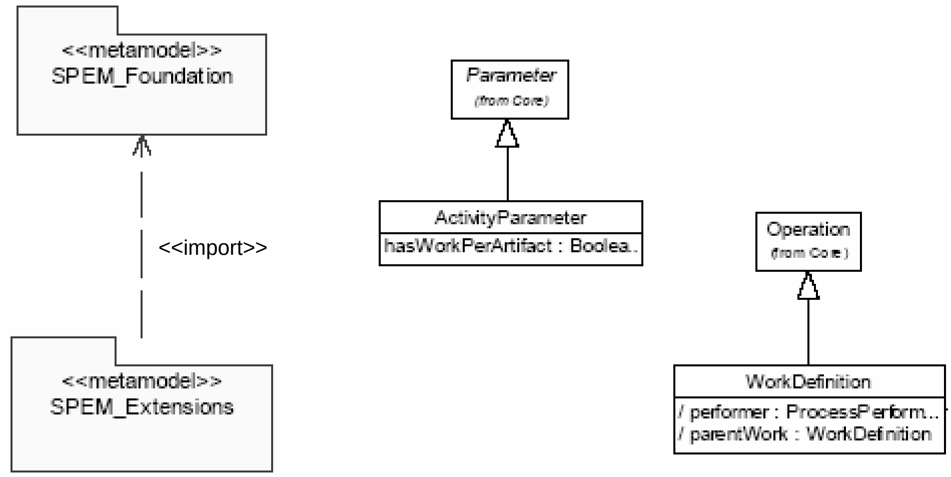


Figure 3.2. Heavyweight extension example: the SPEM1.1 metamodel

Finally, the newly adopted UML2.0 introduces a new extension mechanism through the *Package Merge* relationship. A package merge is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. It is very similar to *Generalization* in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both [OMG 07b]. It is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end.

Package merge is particularly useful in meta-modeling and was extensively used in the definition of the UML metamodel. Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. Examples and more details can be found in the standard specification [OMG 07b]. A discussion on its applicability is addressed in [Zito 06a, Zito 06b].

3.2. Lightweight extension

The Profile mechanism has been specifically defined for providing a lightweight extension to adapt the UML standard for a specific domain. The profiles mechanism is not a first-class extension mechanism i.e., it does not allow for modifying existing metamodels (contrary to the heavyweight extension). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile.

In this approach, the extension is based on the UML customization, in which some of the language's elements are specialized, imposing new restrictions on them. These specializations are defined while respecting the UML metamodel and leaving the original semantics of its elements unchanged. Properties of classes, associations, etc will remain the same, only new constraints will be added to their original definitions and relationships.

To not restrict the profile mechanism only to the UML metamodel but to be applicable to any MOF-defined language, in the latest OMG standard revisions (UML2.0, MOF2, etc.), the *Profiles* package was intentionally defined in the *Infrastructure* [OMG 07a]. It contains mechanisms and concepts that allow metaclasses from existing MOF-instance metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms such as J2EE or .NET or domains such as real-time or business process modeling. Main concepts of this extension mechanism are *stereotypes* (domain-specific class that will extend a metaclass), *tagged values* (in UML1.x, standard metaattributes in UML2.x) and *constraints* (pre/post conditions, invariants, etc).

In UML 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of *stereotypes* and *tagged values*. The UML2.0 Infrastructure and Superstructure specifications have carried this further, by defining it as a specific meta-modeling technique. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages. These changes were mainly inspired by works introduced in [Clark 02], [2U] and [D’Souza 99].

In figure 3.3, we present a very simple yet demonstrative example presented in [Fuentes 04] showing the use of Profiles. Authors also give a very interesting guide on how to build UML profiles. The example is that one wants for instance to add new elements to UML models – say, *weights* and *colours*. Furthermore, one may want to incorporate some particular properties and requirements of such elements, such as the actual *color* of a *coloured* element, the *weight* of a *weighed* element, and a restriction that states that *coloured* associations can only be defined between classes of the same *colour* as that of the association. For the sake of simplicity, we will assume here that only classes and associations can be *coloured*, and that only associations can be *weighed*. The *WeightsAndColours* profile defines these two elements:

First, a *Stereotype* is defined by a name and by the set of metamodel elements it can be attached to. Graphically, stereotypes are defined within boxes, stereotyped «stereotype». In the example, the *WeightsAndColours* UML Profile defines two stereotypes, *Coloured* and *Weighed*, and indicates that both UML classes and associations can be *coloured* (i.e., stereotyped «Colored»), but only associations can have an associated *weight* (i.e., stereotyped «Weighed»). Metamodel elements are indicated by classes stereotyped «metaclass». The notation for an extension is an arrow pointing from a stereotype to the extended class, where the arrowhead is shown as a solid triangle. An Extension may have the same adornments as an ordinary association, but navigability arrows are never shown.

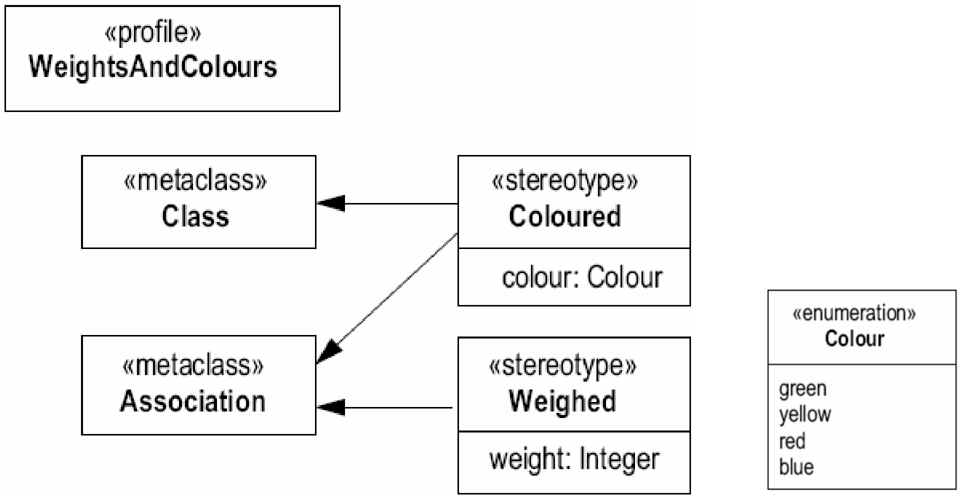


Figure 3.3. Lightweight extension: an example of a profile specification

Second, *Constraints* can be associated to stereotypes, imposing restrictions on the corresponding metamodel elements. Examples of constraints include pre- and post-conditions of operations, invariants, derivation rules for attributes and associations, the body of query operations, etc. In this way a designer can define the properties of a “well-formed” model. For instance, the aforementioned restriction on the colours of the classes joined by a *coloured* association can be expressed by the following OCL constraint:

```
context UML::InfrastructureLibrary::Core::
      Constructs::Association
inv : self.isStereotyped(“Coloured”) implies self.connection->forAll
(isStereotyped(“Coloured”)
   implies (color=self.color)
```

Finally, a *tagged value* is an additional meta-attribute that is attached to a metaclass of the metamodel extended by a Profile. Tagged values have a *name* and a *type*, and are associated to a specific stereotype. In the example, the stereotype «Weighed» has an associated tagged value named “weight”, of type Integer that represents the actual weight of the stereotyped association. Graphically, tagged values are specified as attributes of the class that defines the stereotype.

3.3. Lightweight extension Vs. Heavyweight extension

Each of the aforementioned extension mechanisms has its advantages and disadvantages which may make it, depending on the context and domain, suitable or not for specific modeling purposes. Both of them aim to define a notation and semantics in order to deal with a particular application domain. It is not therefore obvious to decide when to create a new language and when to define a profile. This is what induced some OMG's standards such as SPEM, EDOC and recently, SysML to opt for both mechanisms i.e., a MOF metamodel and a UML Profile [OMG UMLpf].

Defining a tailor-made language will produce a notation and semantics that will perfectly match the concepts and nature of the specific application domain. However, as the new language does not respect the UML semantics, it will not allow leveraging the bunch of UML tools already provided by tool vendors. Nevertheless, recently, with the emergence of some MOF-Based metamodelling environments, repositories and code generator tools such as Vanderbilt's University GME-MOF tool [Emerson 04], the Xactium's XMF-mosaic tool, the Kermeta environment [Muller 05] and the Eclipse's EMF (Eclipse Modeling Framework) [EMF], GEF (Graphical Editing Framework) [GEF] or GMT [GMT] (Generative Modeling Technologies), this is not really considered as an obstacle anymore.

Conversely, UML Profiles may not provide such an elegant and perfectly fitting solution as it may be required for some domains [Peltier 02]. On the other hand, they offer the possibility to reuse UML tools instead of creating new ones from scratch. Another advantage is the number of people already familiar with UML specification and tools. However whether current tools allow the definition and usage of UML Profiles, this is only done at the diagrammatic level, i.e., only graphically. This means that verification of constraints (e.g., in OCL) associated to stereotypes is not yet supported or have to be expressed with some proprietary languages (e.g. the use of the J language in the case of the Objecteering Profile Builder tool [Objecteering]), and consequently well-formed rules can be neither checked nor enforced [Fuentes 04]. The user can therefore never be sure whether or not the system being specified using a given profile is conformant with profile rules.

Regarding the debate Profile versus Metamodel, even the OMG does not give a precise answer. It only states: "*there is no simple answer for when you should create a new metamodel and when you instead should create a new profile*". However, in [Desfray 00], Desfray gives some insights on whether one should define a profile or a new metamodel.

One may opt for heavyweight extension if:

- The target (specific) domain is well defined, and has a unique well accepted main set of concepts
- A model realized under this domain is not subject to be transferred into other domains
- There is no need to combine this domain with other domains

One may opt for lightweight extension if:

- The domain is not subject to consensus, many variations and point of view exist
- Many changes and evolutions may occur
- The domain may be combined with other domains, in an unpredictable way
- Models defined under your domain may be interchanged with other domains

In our proposition we opted for the heavyweight extension mechanism (i.e., first-class extension mechanism). The domain of software process modeling is well defined and there is a consensus on the set of concepts proper to this domain. These concepts were introduced in the previous chapter (cf. section 3.1.). Additionally, software process models once defined, are not meant to be combined with other models from different domains neither to be transferred into other domains. Another motivation for choosing this extension mechanism is that we intended to participate in the OMG's SPEM revision namely, SPEM2.0. In the RFP (Request For Proposal) [OMG 04], it was a mandatory requirement to define a MOF metamodel for the future standard. The heavyweight extension mechanism allows us to define new metaclasses with semantics proper to the software process modeling domain, which are lacking in UML, as well as a notation. Using the lightweight extension mechanism would be more complex since many constraints on UML metaclasses and relationships have to be defined and no real support for expressing them is provided by current UML2.0 tools. Finally, regarding the tooling aspects, the facilities offered with the Eclipse open source projects (i.e., EMF, UML, GMT, etc.) make the task of defining a tool easier since a large community is already familiar with their use and efficient support is ensured. However, we do not exclude the possibility of defining a UML2.0 profile for our proposition in order to deal with the software process modeling within UML2.0 tools.

4. Conclusion

In this chapter we have introduced MDE and its principles. We have also emphasized on how the software process modeling community can take advantage of these principles (i.e., *Abstraction*, *Executability* of *Models*, *Standardization*, *and Metamodelling*) for a better productivity and less complexity in modeling software processes. These principles are taken into account in the next chapter while comparing UML-Based SPMLs. Since our approach is based upon UML, we believed essential to introduce UML's main characteristics and most of all, the extension mechanisms it proposes. In the next chapter, we present a survey on UML-Based SPMLs, which highlights advantages and drawbacks of each approach.

Chapter 4

UML-Based Software Process Modeling Languages

1. Introduction

In this chapter, we present a survey on UML-based software process modeling languages. Before comparing the different propositions, we will first establish a set of requirements, which represent major considerations to be satisfied for the definition of a software process modeling language. These requirements, gathered from well-known works done in the literature, are to be combined with MDE considerations introduced in the previous chapter. The crosscutting between well-established requirements in the community of SPMLs and those needed for an efficient use of the MDE approach will be our basis for comparing the different approaches.

In the following, Section 2 introduces principal SPML requirements, which will be used in the comparison of UML-Based SPMLs presented in Section 3. Section 4 discusses the result of this evaluation and whether the current UML-based SPMLs fulfil requirements we defined or not. Finally, Section 5 concludes this chapter.

2. Requirements for Software Process Modeling Languages

The large number of potential process model users, such as software process engineers, project managers, software engineers, system engineers, software executives, and customer management makes it difficult to establish a universally understood representation format [Curtis 92]. Due to their individual information needs and expertise, these groups place widely diverging demands on a process modeling language. Visual representations, abstraction, and multiple perspectives offer promising techniques for coping with these challenges.

Research on software process modeling identified many requirements for the definition of a SPML [Riddle 89] [Kellner 89] [Curtis 92] [Jaccheri 99]. They vary from facilitating human understanding to providing automated execution support. In the following, we introduce principal ones and we motivate how the ones we selected can be an important criteria while designing a SPML. These requirements will be taken into account while comparing UML-Based SPMLs in section 3.

2.1. Semantic Richness

Semantic richness relates to the SPML ability to express what is actually performed during software development processes, even for most complex situations. It is a very large requirement and encompasses many aspects. Herein we detail each of them.

2.1.1. Process Elements

It relates to the fact that an SPML has to offer the appropriate set of concepts in order to cover the description of all process elements. Most frequently mentioned process model elements are: *Agent*, *Role*, *Activity* (or *Step*) and *Artifact* [Dowson 91], [Humphrey 92], [MacLean 89]. However, there is not a strict agreement on a fixed list of process model elements. In [Conradi 95] for instance, authors add as a crucial element, the notion of *Tool*. The definition of each of these elements was given in the Chapter 2 (cf. section 3.1.).

2.1.2. Activities and Actions Coordination

One important criterion, which is centric to most PMLs, is the mechanisms by which activation, sequencing, iteration and synchronization of activities / actions (steps) is ensured. These mechanisms fall into two categories, *Proactive Control* and *Reactive Control* [Wise 00]. Proactive control is an imperative specification of the order in which activities (steps) are to be executed (direct invocation). Examples of a proactive control are *Control Flow* (i.e., explicitly linking two or more activities/actions) and *Object Flow* (i.e., linking of activities through artifacts. The artifact output of an activity/action may be used as input of other activities/actions). Reactive control is a reactive specification of the conditions or events in response to which activities (steps) are to be executed (indirect invocation). Examples of such control are *Exception* and *Event handling*, *Message receptions*, etc.

2.1.3. Exception Handling

Since exceptions are part of software development process ingredients, we believe that they should be reflected in process models. By placing exceptions in process models, modelers are acknowledging that these elements are part of their overall view and design of the process. It helps to show how they are handled and where they originate from.

2.1.4. Advanced Constructs

Due to the complexity and unpredictable nature of software development processes, advanced constructs are required to express the most complicated situations. A suitable SPML should offer the possibility to express iterations, modeler decisions, synchronizations, *WorkProducts* storage and retrieval, communication between agents, tool invocations, etc. These advanced constructs could also help in facilitating the automation of the process.

2.2. Understandability

SPMLs and process models cannot be used if they cannot be understood. This is what makes this objective so crucial. The large number of actors participating in the development process puts serious constraints on the format that should be used to express process models. Some of them would prefer graphical representations while other would find it more appropriate to handle code, to test it in order to directly get feedbacks on the modeled process. The success of UML and the involvement of people with no computer science background in the software realization confirmed the efficiency of graphical representation in augmenting understandability and in easing communication between people. Thus, this is considered as a crucial point to take into account while designing a SPML.

Another important aspect to consider and which can be added to the understandability requirement, is the support of different perspectives /views on the process. Here the term view is employed informally and just means, a representation of the process according to certain aspects. Some of them were introduced in section 2.2.4 of Chapter 2 (i.e., *Activity view, Product view, Role View,* etc).

2.3. Precision

By precision, we mean here the degree of details (i.e., granularity) used to describe software activities. One would argue that abstract process models promote understandability. This is true but only if they don't aim to be executed. Thus, a compromise has to be found in order to combine precision and understandability, which is, in our view, a quite difficult exercise.

2.4. Executability

Nowadays, companies are increasingly driven by the need to extensively automate all parts participating in the production of software, including the development process itself. However, this goal cannot be reached if the SPML does not provide constructs with operational semantics as well as the execution support for these constructs.

2.5. Modularization

Modularization is about to be able to combine different chunks of processes in order to build a new one. This means that the SPML should provide such concepts that allow composing already defined processes easily and without modifying them. This would be a beneficial advantage especially, that nowadays we assist to the emergence of new way of working such as outsourcing, contracting-out, etc., which imposes more flexibility on SPMLs in order to consider process composition constraints.

The above-cited requirements represent main SPML requirements that we believe essential for designing a SPML. Added to MDE considerations introduced in the previous chapter, these requirements will be used as a basis for comparing the different UML-Based SPMLs presented in the next section.

3. Comparing UML-Based Software Process Modeling Languages

In this section, we compare UML-Based Languages for Software Process Modeling according to the requirements introduced in the previous section. We also take into account if, whether or not, the language fulfills some of the MDE considerations addressed in Chapter 3. i.e., does the language provides a metamodel, is there any means to execute its model instances, is it based on a standard formalism (in this case it is always true since all approaches here are UML-Based), etc. At the end of this comparison, a table summarizing all aspects of languages discussed here is presented.

3.1. SPEM1.1

SPEM1.1 (Software Process Engineering Metamodel) is the OMG's standard for software process modeling. It was adopted by January 2005 [OMG 05a], and is the first revision of the standard which was initially issued by end of 2002 [OMG 02]. During this thesis, another revision (i.e. SPEM2.0) of the standard was requested through an RFP (Request For Proposal) [OMG 04] and when writing this document, finalization tasks of SPEM2.0 were underway. In the SPEM2.0 revision, LIP6 was heavily involved in the standardization process. SPEM2.0. is presented in detail in the next section (c.f., 3.2.)

SPEM1.1 introduces common concepts and a modelling structure to construct models of software development processes. It uses some basic modelling concepts from UML1.4 to describe rules, constraints, vocabulary, and notation to be used in defining process models [OMG 01]. It comes in form of a MOF1.3-compliant metamodel and a UML1.4 profile. The metamodel is defined as an extension of a subset of UML1.4, expressed in the *SPEM_Foundation* package. The *SPEM_Extensions* package which extends the *SPEM_Foundation* package, adds the constructs and semantics required for software process engineering. It owns five packages; each package addresses a specific concern of the software process definition.

The building block of the SPEM metamodel is the *Process Structure* package (cf. figure 4.1.). It defines the main structural elements from which a process description may be constructed.

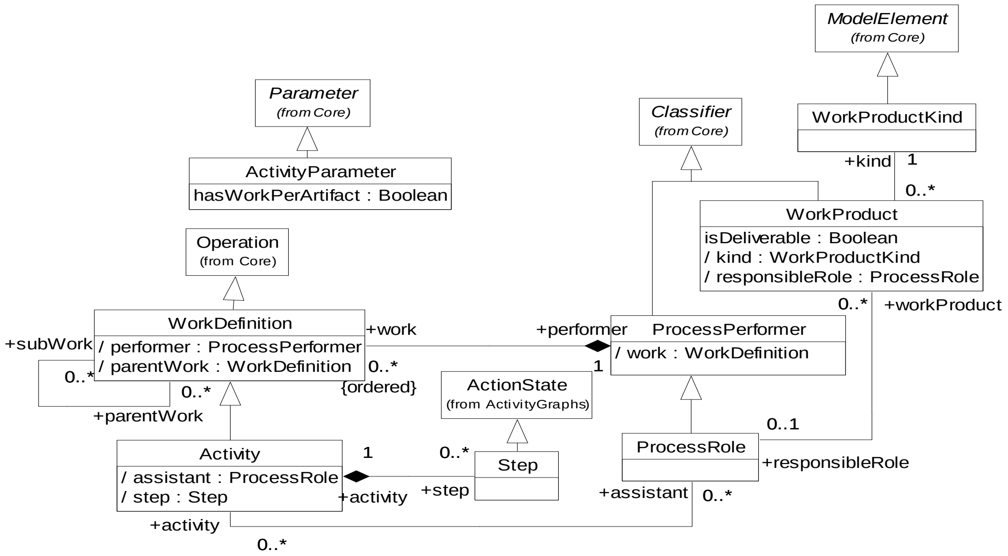


Figure 4.1. The Process Structure package, the SPEM1.1 metamodel core for process definitions.

3.1.1. SPEM1.1 Evaluation

In the following we evaluate SPEM1.1 according to the SPML requirements.

3.1.1.1. Semantic Richness

As we saw in the previous section, this requirement covers many aspects. We detail each of them in the context of SPEM1.1.

Process Elements

The notion of **Activity** in SPEM1.1 is given through the *Activity* metaclass, which is the main subclass of *WorkDefinition*. It describes a piece of work performed by one *ProcessRole* and may consist of atomic elements called *Steps*.

The notion of **Product** corresponds to a *WorkProduct* in SPEM and is anything produced, consumed, or modified by a process. It describes one class of artifacts produced in a process and has a *WorkProductKind* that describes a category of artifact, such as Text Document, UML Model, Code Library, etc. Finally, the notion of **Role** in SPEM, corresponds to *ProcessRole*, which is a subclass of *ProcessPerformer* and defines responsibilities and roles over specific *WorkProducts* and *Activities*.

Although SPEM1.1 defines the notion of *ProcessRole* (Role), it does not provide the one of ***Human*** or ***Agent*** who may undertake this role. Moreover, there is no concept in the standard equivalent to the notion of ***Tool***. Besides, there is no support or concept related to human interactions

Activities and Actions Coordination

In SPEM1.1 the ability to orchestrate process Activities and Steps is only provided by means of a proactive control thanks to the *Precedes* dependency. Kinds of precedence were: *start-start*, *finish-start* or *finish-finish*. The *start-finish* precedence is lacking. Regarding the reactive control, SPEM1.1 does not provide any concept to deal with this aspect.

Exception Handling

Exception handling is not addressed in SPEM1.1.

Advanced Constructs

The standard does not define concretely concepts such as loops or conditionals. Rather, it defines the notion of *Phase* and *Iteration*. A *Phase* is a specialization of *WorkDefinition* such that its precondition defines the phase entry criteria and its goal (often called a "milestone") defines the phase exit criteria. However it does not allow expressing a condition to be evaluated as a result of an activity or step execution (i.e., at lower level, since a Phase may contain Activities). An *Iteration* is a composite *WorkDefinition* with a minor milestone and expresses the need to iterate the work until a certain goal is reached. This goal is also expressed thanks to post conditions. No concepts for easing the communication between agents are provided, no constructs for modeling tool invocations or WorkProducts storing and retrieval are defined by SPEM1.1.

3.1.1.2. Understandability

SPEM1.1 uses UML notations and diagrams. This is considered as a serious advantage as UML has attractive features: it is standard, graphical, intuitive, and easy to be understood. Besides, a wide community of software developers is familiar with UML and uses a UML case tool environment. UML being so popular and widely used, SPEM has an important competitive advantage compared to any specialized PML [Di Nitto 02].

Since understandability is considered as a decisive criterion in adopting a modeling language, the above-cited arguments comforted us in using UML as a basis of our SPML. However, for the approaches discussed here, we will see that the choice of the UML concepts to use, to extend, the set of diagrams to employ can be very decisive in improving or not the understandability of the SPML.

3.1.1.3. Precision

SPEM1.1 proposes different concepts for organizing the process description into a hierarchy. These concepts range from *Phase*, *Lifecycle*, *Iteration*, and *WorkDefinition* to *Activity*, and *Step*, which are considered as the lower level constructs for describing a process. However the *Step* element is defined only as a means of describing guidelines for performing the activity. There is no way to express what are the inputs/outputs of the step, who is the role responsible of this step neither their constraints. All these concerns are expressed at the activity level which makes it the effective lower level for describing process's tasks in SPEM1.1. Indeed, In SPEM1.1, a *WorkProduct* inherits from the UML1.4 *Classifier* and is used as a parameter into or from *Activities* (*WorkDefinition* in general). We can't know which steps of the *Activity* are going to act on *WorkProducts* nor responsible roles of these *Steps*. The standard also proposes a notation for each of the above-mentioned concepts (i.e., *Phase, Iteration,* etc.) except for the *Step* element. Moreover, in all examples introduced in the specification, there is no one who deals with *Steps*.

We believe that for a better technology transfer, for a good comprehension of process performers about the work to be accomplished and in order to go toward a larger automation of process parts, more precise process models are required.

3.1.1.4. Executability

The automation of process model executions requires that the SPML provides such concepts with an execution semantics that would allow their mapping toward some execution formalisms and languages. SPEM1.1 provides as atomic actions of a development activity, the concept of *Step,* which only represents the name of the action that developer has to perform (e.g., Step x: "Check model consistency"). In the standard, the only reference to the *Step* element is "*An Activity may consist of atomic elements called: Steps*" [OMG 02]. Obviously, this is insufficient. A *Step* inherits from UML1.4 *ActionState*. "*An action state represents the execution of an atomic action, typically the invocation of an operation*" [OMG 04]. But, UML1.4 does not explicitly specify, neither parameters of the invocation action (i.e., name and value) nor their types as it is done with *Actions* in UML2.0. Moreover, SPEM1.1 adds the constraint that a *Step* has no associated action.

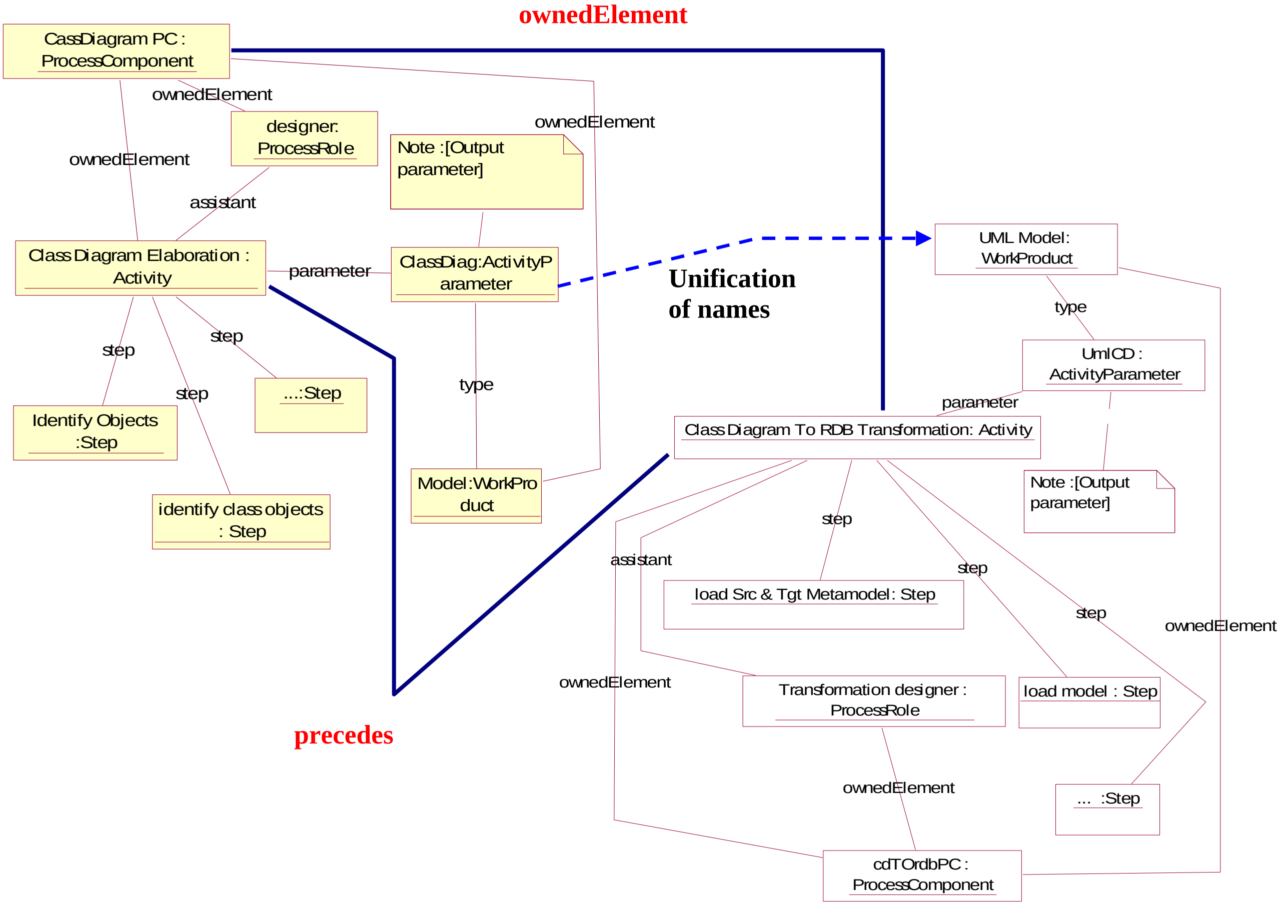
In SPEM1.1, the use of the *Step* element could help for process description but it is so far of its execution. We agree that execution of process models was outside the scope of SPEM1.1. However, we hardly believe that it should provide concepts that enable the specification of executable action semantics within process models.

UML2.0 offers this possibility thanks to the *Actions* packages. It gives precise execution semantics to actions, by defining their effect as well as their typed inputs and outputs. This may help in mapping them into executable actions in some well-known OO languages such as Java or C++. This also comforted us in our choice of reusing some constructs (i.e., *Activities* and *Actions*) of the UML2.0 Superstructure in our SPML.

3.1.1.5. Modularization

One of the major lacks of SPEM1.1 is *ProcessComponent* compositions, which is supposed to be the mechanism for process compositions. A *ProcessComponent* is a chunk of process description that is internally consistent and may be reused with other *ProcessComponents* to assemble a complete process. However, developers who want to combine two or more *ProcessComponents* in order to get one coherent process, have to carry out a *unification procedure*. Indeed, to combine for instance two *ProcessComponents* P1 and P2, at least the output *WorkProducts* from P1 must be unified i.e., made identical with the *WorkProducts* inputs to P2. Other elements may possibly be unified in addition, such as *ProcessRoles*.

Let's have for example two Process Components PC1 and PC2 (see figure 4.2.). PC1 is in charge of realizing a UML class diagram. PC2 has to transform a UML class diagram to a relational database diagram. These two processes were specified separately, so *WorkProducts* and *Roles* might have different names. If a process modeler decides to compose these two process components, she/he will have to unify (rename) *WorkProducts* output from PC1 (i.e., ClassD) in order to be identical with *WorkProducts* inputs of PC2 (i.e., UmlCD). Likewise, she/he has also to explicitly link activities from PC2 within PC1.



PC1: Class Diagram Process Component **PC2: Class DiagramToRDBTransformation Process Component**

Figure 4.2. Process Component compositions in SPEM1.1

Composition of *ProcessComponents* can be fully automated only if they originate from a common family (i.e., an agreement on *WorkProduct* and *Roles* names) so that the unification is capable of being automated. Otherwise, the unification would involve human intervention that normally would consist of some re-writing of the elements, and possibly associated elements, to be unified. This could be manageable in case of the combination of two simple *ProcessComponents*. However in case of complex *ProcessComponents*, it becomes increasingly difficult.

3.1.2. Discussion

To summarize, SPEM1.1 presents the advantage of being based upon UML, which makes it easy to understand and a good candidate for a large adoption since many people are familiar with the unified modeling language. However the standard has had a limited success within the industry. One of the obstacles was that the standard comprised many ambiguities. As an example, let's consider the concept of *ProcessPerformer*. The standard defines the *ProcessPerformer* as a performer for a set of *WorkDefinitions*. It also states that *ProcessPerformer* represents abstractly the “whole process” or one of its components. Definitively, we can clearly note that this definition is confusing. One obvious question would be: what is the practical use of a *ProcessPerformer*? Is it used as a *container* for *WorkDefinitions* or as a *role,* responsible for specific activities? In the latter case, what is the difference with the *ProcessRole* concept? We believe that a container of *WorkDefinitions* and *roles* are totally two separate concepts that should be expressed separately. Another example is the relationship between a *Discipline* and a *Process*. Since both concepts inherit the *Namespace* and *ModelElement* UML1.4 elements, this makes that a *Process* can be composed of *Disciplines* (through the composition between a *Namespace* and a *ModelElement*) but also allows to have process models where a *Discipline* may be composed by *Processes*, which is in contradiction with the semantics given to the *Discipline* concept by SPEM1.1. Thus, this relationship has to be constrained for more coherence. In [Bendraou 05], we identified many of these ambiguities and other inconsistencies with some solutions are proposed in [Combemale 06].

Another obstacle for the adoption of SPEM1.1 was that process models were not executable and do not provide concepts with execution semantics. SPEM1.1 process models were contemplative models.

Regarding the tooling aspects, only few implementations of the standard are proposed. We can cite the Rational Process Workbench (RPW) from Rational [RPW], IRIS Suite from Osellus [Osellus], and SPEM Profile from Objecteering [Objecteering]. However, we evaluated these tools and no one of these implementations is in 100% conformity with the standard. Moreover, each of them proposes its own formalism for process model persistency. Thus no model exchanges are possible between the different tools.

3.2. SPEM2.0

SPEM2.0 is the revision of SPEM1.1 and when writing this document, the specification where under finalization (FTF: Finalization Task Force). SPEM2.0 aims at providing organizations with means to define a conceptual framework offering the necessary concepts for modeling, interchanging, documenting, managing and presenting their development methods and processes [OMG 07c]. Besides providing a standard way for representing organization’s processes and expertise, SPEM2.0 comes with a new attractive vision. That latter consists in separating all the aspects, contents and materials related to a software development methodology from their possible instantiation in a particular process. Thus, to fully exploit this framework, the first step would be to define all the phases, activities, artifacts, roles, guidance, tools, and so on, that may compose a methodology and then, to pick, according to the situation or process context, the appropriate method contents to use within a process definition.

SPEM2.0 comes in form of MOF-compliant metamodel that reuses UML2.0 Infrastructure [OMG 07a] and UML2.0 Diagram Interchange specifications [OMG 06d]. It reuses from the UML Infrastructure basic concepts such as *Classifier* and *Package*. No concept from the UML2.0 Superstructure [OMG 07b] is reused. The standard comes also in form of a UML Profile where each element from the SPEM2.0 metamodel is defined as a stereotype in UML2.0 Superstructure. The metamodel is composed of seven packages linked with the "merge" mechanism (cf [OMG 07a], §11.9.3), each package dealing with a specific aspect (cf. fig. 4.3.).

The *Core* package introduces classes and abstractions that build the foundation for all other metamodel packages. The building block of this package is the *WorkDefinition* class, which generalizes any work within SPEM2.0. The *Process*

Structure package defines elements for representing basic process models in terms of a flow of *Activities* with their *WorkProduct Uses* and *Roles Uses* (figure 4.4.). However, the possibility to textually document these elements (i.e., add properties describing the element) is not provided in this package but in the *Managed Content* package. That latter provides concepts for managing the textual description of process elements. Examples of such concepts are the *Content Description* class and the *Guidance* class. The *Method Content* package defines core concepts for specifying basic method contents such as *Roles*, *Tasks* and *WorkProducts*. The *Process with Method* package defines the set of elements required for integrating processes defined by means of *Process Structure* package concepts with instances of *Method Content* package concepts. The *Method Plugin* package provides mechanisms for managing and reusing libraries of method contents and processes. This is ensured thanks to the *Method Plugin* and *Method Library* concepts. Finally, *Process Behavior* package provides a way to link SPEM2.0 process elements with external behavior models such as UML2.0 Activity Diagrams or BPMN (Business Process Modeling Notation) models [OMG 06a].

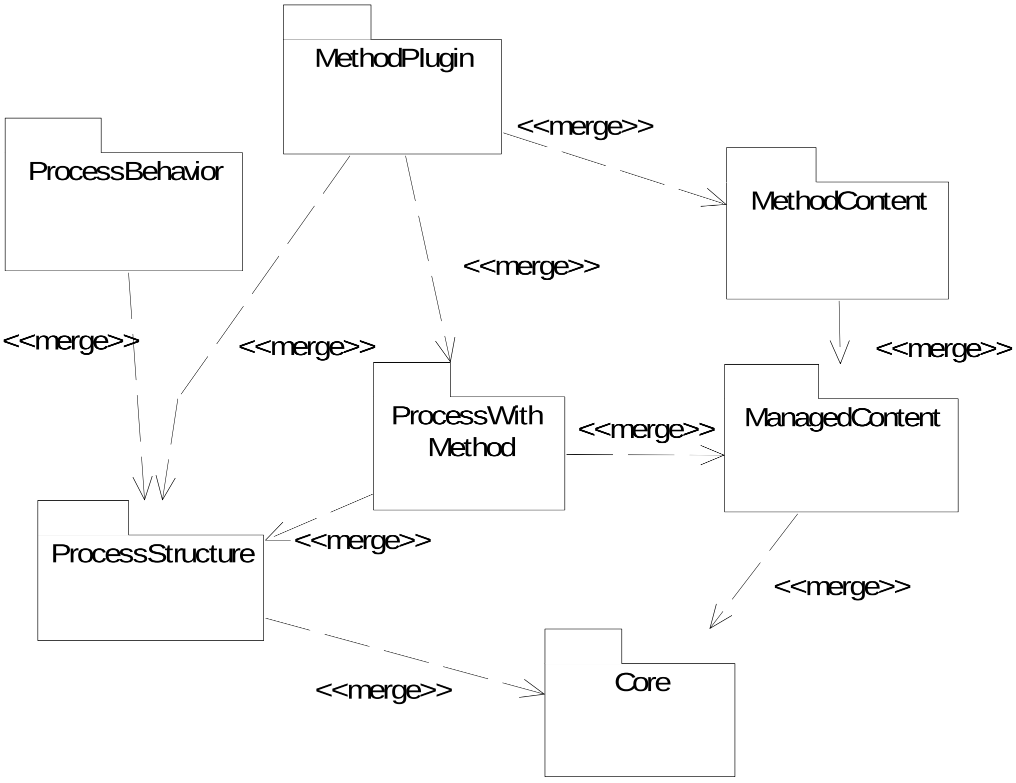


Figure 4.3. SPEM2.0 Metamodel package hierarchies

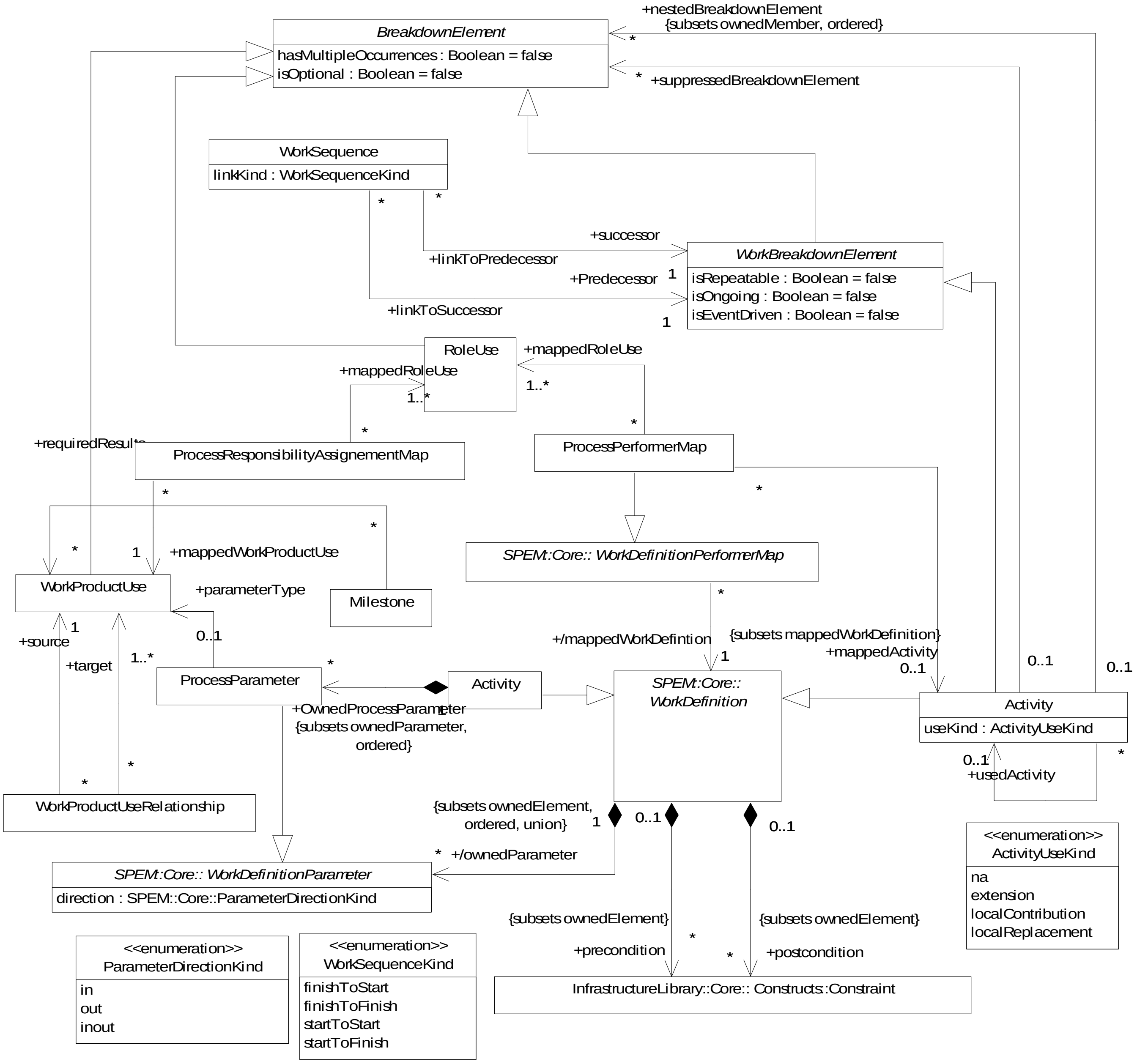


Figure 4.4. SPEM2.0 Process Structure Package

3.2.1. SPEM2.0 Evaluation

In the following we evaluate SPEM2.0 according to the SPML requirements.

3.2.1.1. Semantic Richness

Before to go through the different aspects of the semantic richness requirement, we need first to introduce the implementation compliance points defined by the standard.

To implement the specification, SPEM2.0 defines three compliance points. The first one called "*SPEM Complete*" is dedicated for case tool providers that want to support the description of large-scale method libraries as well as the definition of process models that may reuse the method library contents. It contains all the packages introduced above (cf. Figure 4.3.). The second compliance point is the "*SPEM Process with Behavior and Content*" and is dedicated for tool providers who are just interested in providing concepts for describing process models without referring to a particular method. This compliance point was especially designed for the *Agile* community which prefers simple and brief process description to method hand-books and guidelines. In this compliance point, there is no possibility to reuse method content descriptions. It is composed of four packages: the *Process Structure* (cf. Figure 4.4.), the *Core*, the *Process Behavior* and the *Managed Content* packages. The last compliance point is the *"SPEM Method Content"* and it is recommended for implementers who primarily focus on managing the documentation of descriptions of development methods, techniques, and best practices. It comprises the *Method Content*, *Managed Content* and *Core* packages.

So depending on the compliance point implemented, the tool may provide or omit some of the SPEM2.0 constructs.

Process Elements

As introduced previously, SPEM2.0 aims at giving the possibility to process modellers to separate a method description from the possible instantiation of its contents (*Role Definitions*, *Task Definitions*, *WorkProduct Definitions*, etc.) in a particular process (*Role Uses*, *Activities*, *WorkProduct Uses*, etc.). What makes the thing complex is that distinct concepts, which semantically have the same meaning (use) are used differently depending if they are part of a process model or of a method description.

For instance, if you are using a tool implementing the *"SPEM Process with Behavior and Content"* compliance point, you will refer to artifacts used by process's activities as *"WorkProduct Uses"* and to process's roles as *"Role Uses"*. A *"WorkProduct Use*" is defined by the specification as an input and/or output type for an *Activity* [OMG 07c]. However, when using a tool implementing the *"SPEM Method Content"* compliance point and you are describing a method, you will refer to artifacts produced or consumed by a method's *Task Definitions* as *"WorkProduct Definition"*. A *WorkProduct Definition* is introduced as tangible work products consumed, produced, or modified by *Tasks*. Finally, if you are using *"SPEM Complete"*, you can use both *WorkProduct Use* and *WorkDefinition*. The *WorkDefinition* will used for documenting the artifact used by your method and the *WorkProduct Us*e will be used in your process model as reference (pointer) to the *WorkProduct Definition* given in the instantiated method instead of re-describing once again the artifact within the process model.

Thus, as we can notice process elements that you may use depends on the compliance point supported. Basic process elements are ensured thanks to concepts introduced in the previous paragraph however, as in SPEM1.1, the notion of ***Agent*** is lacking and the notion of ***Tool Definition*** is only provided the *"SPEM Complete"* and *"SPEM Method Content"* compliance points.

Activities and Actions Coordination

In SPEM2.0, no reactive control is provided. Proactive control is ensured thanks to the *WorkSequence* element, which allows orchestrating the different *Work Breakdown Elements* (*Task Definition*, *Activity, and Step*) defined by the specification. The *WorkSequenceKind* property helps in defining the kind of dependency between two *Work Breakdown Elements,* which may be *start-start*, *finish-start* or *finish-finish* and finally the *start-finish*, which was lacking in SPEM1.1

Exception Handling

Exception handling is not addressed in SPEM2.0.

Advanced Constructs

SPEM2.0 does not explicitly define concepts such as *Iteration*, *Phase*, *Lifecycle*, etc. which would be too restrictive or method specific. Instead, during the standardization process, partners (including LIP6) focused on adding a more flexible means that would allow adding new kinds of process elements through the *ExtensibleElement* abstract metaclass defined in the *Core* package(figure 4.5.). This mechanism was inspired by Odell's *"Power Types"* work [Odell 94].

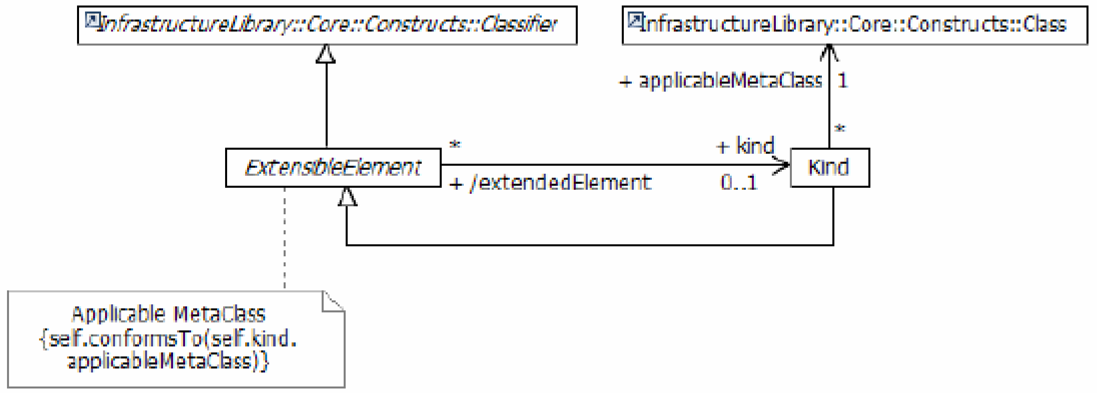

InfrastructureLibrary::Core::Constructs::Classifier
InfrastructureLibrary::Core::Constructs::Class
+ applicableMetaClass
1
*
ExtensibleElement
*
+ kind
Kind
+ /extendedElement
0..1
Applicable MetaClass
{self.conformsTo(self.kind.applicableMetaClass)}

Figure 4.5. The SPEM2.0 Extensible Element construct

An *ExtensibleElement* is an abstract generalization that represents any SPEM 2.0 class for which it is possible to assign a *Kind* to its instances expressing a user-defined qualification. Every SPEM 2.0 class that allows such a qualification derives directly or indirectly from *ExtensibleElement*. Thus, it is for instance possible to define new *Work Breakdown Elements* such as *Phase*, *Discipline* in the context of RUP but also a *Sprint* in the context of Scrum agile processes. The specification does not define constructs for expressing loops or conditionals, tool invocations or agent communications.

3.2.1.2. Understandability

Regarding understandability, we have to admit that the standard is very complex and hard to tackle. We have to deal with many compliance points, which may turn very confusing for non-expert of SPEM2.0. The specification defines seven packages with seventy-five metaclasses. Moreover, SPEM2.0 does not propose any behavioural model for describing the workflow of the process but proposes rather a kind of proxy classes in order to link the process description with some external behavioural models defined in other formalisms such as BPMN for instance, which is inconceivable. We will detail this point in the executability aspect presented below.

Finally, in the RFP [OMG 04], it was fixed as a mandatory requirement, the reuse of UML2.0 Superstructure *Activity* and *Action* constructs while defining the SPEM2.0 metamodel. Unfortunately, this was not taken into account by the principal submission.

3.2.1.3. Precision

When using the *"SPEM Process with Behavior and Content"* compliance point (i.e., to describe the process undependably of any method), the *Activity* is the only concept for describing the hierarchy of the process model. It can represent the whole process as well as atomic actions within the process. *Activities* relate to *Work Product Use* instances via instances of the *Process Parameter* class and *Role Use* instances via *Process Performer Map* instances.

When using the *"SPEM Method Content"* or *"SPEM Complete"* compliance points it is possible to describe the method library in terms of *Tasks* that it may contain, *Steps* of each task, *Process Components* (reusable units of work), *Roles*, *Workproducts*, etc. which represents a quite complete set.

3.2.1.4. Executability

Even if process enactment was among the principal requirements when the SPEM2.0 RFP was issued [12], the final adopted specification does not address the enactment issue. Nevertheless, it clearly suggests two possible ways of enacting SPEM2.0 process models. In the following, we introduce them, we present the concepts that are supposed to be used in order to enact SPEM2.0 process models and we give some criticisms on the feasibility or not of each approach.

Mapping the SPEM2.0 Processes Models into Project Plans

In this first approach the standard proposes to map SPEM2.0 processes into project plans by means of project planning and enactment systems such as IBM Rational Portfolio Manager [RPM] or Microsoft Project [MSProject]. SPEM2.0 processes defined using breakdown structures - i.e., *Activity*, *Role Use* and *WorkProduct Use* from the *Process Structure* package - offer key attributes that provide the project planner with the right guidance to make process instantiation decisions. Examples of such attributes are the *hasMultipleOccurrence* attribute that indicates for instance that an *Activity* or a *WorkProduct* will be mapped to multiple plan *Activities* - respectively multiple plan *WorkProducts* (multiple occurrences of the *Activity/WorkProduct* are needed). An *isRepeatable* flag for an *Activity* indicates that the *Activity* has to be iterated many times before to get the expected result. An *isOptional* attribute indicates whether the *Activity* can be skipped out or not, for example due to a delay in the initial schedule.

Once SPEM2.0 processes mapped to project plans, plans can be instantiated by means of planning tools and concrete resources can then be affected. However, whether this approach may be very useful for project planning it is not considered as process enactment. It is still necessary to affect duration to tasks, persons to roles in order to get at the end an estimation of the development process period and resources needed for its realization. These plans are used by project manager in order to estimate if the process will be in schedule or not, if more persons need to be affected to process tasks, etc. There is no support for process execution, no automatic task affectations to responsible roles, no automatic routing of artifacts, no automatic control on work product states after each activity, no means to support agent and team communications and so on.

Besides the fact that this approach does not provide concrete enactment support, it presents a major lack which is its tight dependence to the project planning tool. Another aspect that has to be taken into account is the impact of modifying or adding information within the planning tool and how this modification will be reflected / traced-up to the SPEM2.0 process model. Finally, process modelers have to deal with the compatibility of process definition file format of the planning tool.

Linking SPEM2.0 process elements with external behavior formalisms

The SPEM2.0 does not provide any concepts or formalism for modeling precise process behavior models or execution. Rather, claiming for more flexibility, SPEM2.0 provides through the *Process Behavior* package a way to link SPEM2.0 process elements with external behavior models. The goal behind is not to restrict or to impose a specific behavior model but to give the process modeler the option to choose the one that fits best its needs. A SPEM2.0 *Activity* can for instance be linked with a BPMN diagram [OMG 06a] in order to represent in more details the activity's steps, control flows, etc. Then, a BPMN execution engine has to be provided or a mapping towards process orchestration language such as BPEL (Business Process Execution Language) [WSBPEL 07] has to be carried out in order to reuse BPEL execution engines. In addition, a *WorkProduct* can for instance be linked to a UML state diagram in order to model possible *WorkProduct's* states and transitions that can make this *WorkProduct* move from one state into another. Here again, a state machine engine has to be integrated to the process execution engine. SPEM2.0 defines a kind of proxy classes i.e., *Activity_ext*, *ControlFlow_ext*, *Transition_ext* and *State_ext* in order to link between SPEM2.0 process elements i.e., *WorkProductUse, WorkDefinition, RoleUse, Activity, WorkSequence* and external behavior model elements. It is up to the process modeler to link each process element with its equivalent in the behavior model. Since a single behavior model may not be expressive enough to represent all the behavioral aspects of the process, several behavior models can be combined.

Whether this approach may provide flexibility in representing behavioral aspects of SPEM2.0 processes, it presents some lacks. The first one is that the standard is not very clear on how the linking of process elements with behavioral models is realized. It just provides proxy classes that make reference (point) to other elements in an external behavioral model. SPEM2.0 supposes that this task is tool implementer's responsibility. Tool implementers have to define a specific behavioral model that has to be automatically generated from the SPEM2.0 process model. This is already the case in the free EPF (Eclipse Process Framework) tool [EPF], which is meant to be the implementation of SPEM2.0. In EPF, a kind of a proprietary activity diagram is partially generated from a process definition. That latter can be refined in order to provide more details on the process activities and their coordination (control flows). However no execution is provided. The second lack is that the mapping from SPEM2.0 process elements into a specific behavioral model can be done differently from one organization to another depending on process modeler interpretation. Thus, a standardization effort may be required in order to harmonize mapping rules between SPEM2.0 concepts and a specific behavior model such as BPEL for instance. The third lack, which tightly relates to the previous one, is that more often concepts in behavior models are richer than in SPEM2.0. This is because behavior modeling and execution languages provide additional concepts related to the technical support and execution of processes while SPEM2.0 concentrates on the "business concerns" of the software development process or methodology (i.e., *Roles, Activities, Guidance,* etc.). Consequently, a full executable code generation from SPEM2.0 is not possible which may impose some refinement steps in behavior models before they can be enacted. This in its turn poses the problem of traceability and how these refinements (changes) can be reflected in the initial SPEM2.0 process model.

3.2.1.5. Modularization

SPEM2.0 provides different mechanisms for reusing, extending and customizing process models and methods. At the *"SPEM Process with Behavior and Content"* compliance point, extension of process models is ensured thanks to the *Activity*'s

"Activity Use Kind" property (enumeration). Depending on the value of the property, a process's activity can 1) extend an activity from another process; 2) can be extended by another process's activity or 3) completely replaces an activity in another existing process.

At the *"SPEM Method Content"* or *"SPEM Complete"* compliance points, the specification proposes mechanisms such as *Variability Element* and *Process Component*. The former allows not only for extending process's activities as in the *"SPEM Process with Behavior and Content"* compliance point but also to any metaclass inheriting the *Variability Element* abstract metaclass. This would allow that a process model or contents of one method redefine, reuse or replace another method's contents or process models. The detail of this mechanism is given in more detail in [OMG 07c]. The latter, i.e., *Process Component*, is a means to define a kind of reusable black boxes of processes identified by their ports (i.e., *Workproducts* inputs and outputs of the *Process Component*). Finally, the concept of *Method Plugin* is introduced. It defines a granularity level for the modularization and organization of method contents and processes. A *Method Plugin* can extend many other *Method Plugins* and it can be extended by many *Method Plugins*.

3.2.2. Discussion

The main advance in the SPEM2.0 specification is the proposition of a clear separation between the contents of a method of their possible use within a specific process. However, this was not to simplify things since many concepts and mechanisms were introduced in order to allow method contents to be reused in process models. Added to extension mechanisms, compliance points, the notion of *Method Plugins*, the specification turns out very complex and hard to understand. This complexity comes from the fact that the SPEM2.0 proposition (submission) took as a basis the IBM's UMA (Unified Method Architecture) method, which in its turn is the result of combining three methods: the RUP (Rational Unified Process) [Kruchten 03], the IBM Global Services Method [IBM 97] and the Summit Ascendant Method.

Regarding executability, we saw that SPEM2.0 does provide neither concepts nor formalisms for executing process models. However, it proposes two possible approaches for their execution. We exposed these approaches and we demonstrated their limits.

For modularization aspects, the standard proposes powerful mechanisms for extending process models and methods, which requires extensive implementation efforts in order to respect the semantics of all the proposed extension mechanisms and to make sure that they will not overlap.

Finally, for the tooling support, an open source project called EPF (Eclipse Process Framework) [EPF], which was initially lunched for supporting the IBM's UMA method, is on the way to be fully compatible with SPEM2.0. A commercial version of this tool exists: the Rational Method Composer (RMC) tool [RMC]. Objecteering also proposes a commercial tool on top of EPF and Microsoft Project called PRO3 [Objecteering]. For tool vendors who aim to implement the SPEM2.0 profile they will have to face a considerable obstacle, which is the constraining, using OCL, of the UML metamodel in order to respect the SPEM2.0 metamodel semantics. Indeed, the specification defined the profile but intentionally left the writing of OCL rules up to the profile implementers. The argument was that the semantic is already defined in the SPEM (MOF) metamodel.

3.3. Di Nitto et al. approach

Di Nitto's et al. approach was proposed at the ICSE'02 (International Conference of Software Engineering) [Di Nitto 02]. The approach aims at assessing the possibility of employing a subset of UML1.3 [OMG 00b] as an executable PML. It comprises two main phases. The first one consists in describing process aspects using UML diagrams. The second phase consists in translating this UML diagrams into code that can be enacted by the team's events-based workflow engine called OPSS (ORCHESTRA Process Support System) [Cugola 01]. This team has a good experience in the domain of software process modeling and was among the pioneers in proposing SPMLs [Bandinelli 93] [Bandinelli 95] and [Arlow 97]. They started by a Petri-nets-based SPML named SPADE [Bandinelli 96] and quickly they realised the necessity to raise the abstraction level of SPMLs for a larger adoption. Their experience, their arguments that UML represents a good candidate for a high-level SPML justifies our choice of using UML as a building block of our approach.

3.3.1. Di Nitto et al. approach Evaluation

In the following, we evaluate this approach according to considerations we set earlier in this chapter.

3.3.1.1. Semantic Richness

This approach proposes to use UML1.3 diagrams as a high-level modeling language. There is no extension to the UML1.3 metamodel, no stereotyping or new concepts introduced. These diagrams are then translated into Java code in order to be enacted by the OPSS engine. During the translation, a set of predefined classes used by the OPSS are taken as a basis and extended in order to generate the final Java code. These predefined classes represent basic process constituents such as *Activity*, *Artifact*, *Agent*, etc (cf. figure 4.6.).

The UML diagrams used in this approach are:

- The activity diagram for modeling the flow of work;

- Class diagrams to associate concepts belonging to the level of the process description with concepts that are part of the OPSS. In class diagrams, the UML inheritance (generalisation/specialisation) relationship is used to specialise the set of predefined classes used by the OPSS. These predefined classes, which are given in figure 4.6., can then be specialized by process modeller's classes in order to adapt the predefined class diagram to its specific needs. A process modeler willing to use OPSS has to start defining his/her own activity types, agent types, etc. by specializing the existing classes. This means adding the set of roles classes involved in the process such as *Designer*, *Analyst*, etc., specific artifacts classes such as *Requirements*, *Design document*, and process activities such as *Test Code*, *Edit Report*, and so on.Via specialization the modeler can modify the default values of the attributes of the base classes, or introduce new operations.

- For each OPSS class, a state diagram is associated in order to describe the lifecycle of each entity. A precise and complete definition of these state machines is curial for process execution, since that they encapsulate the process business rules.

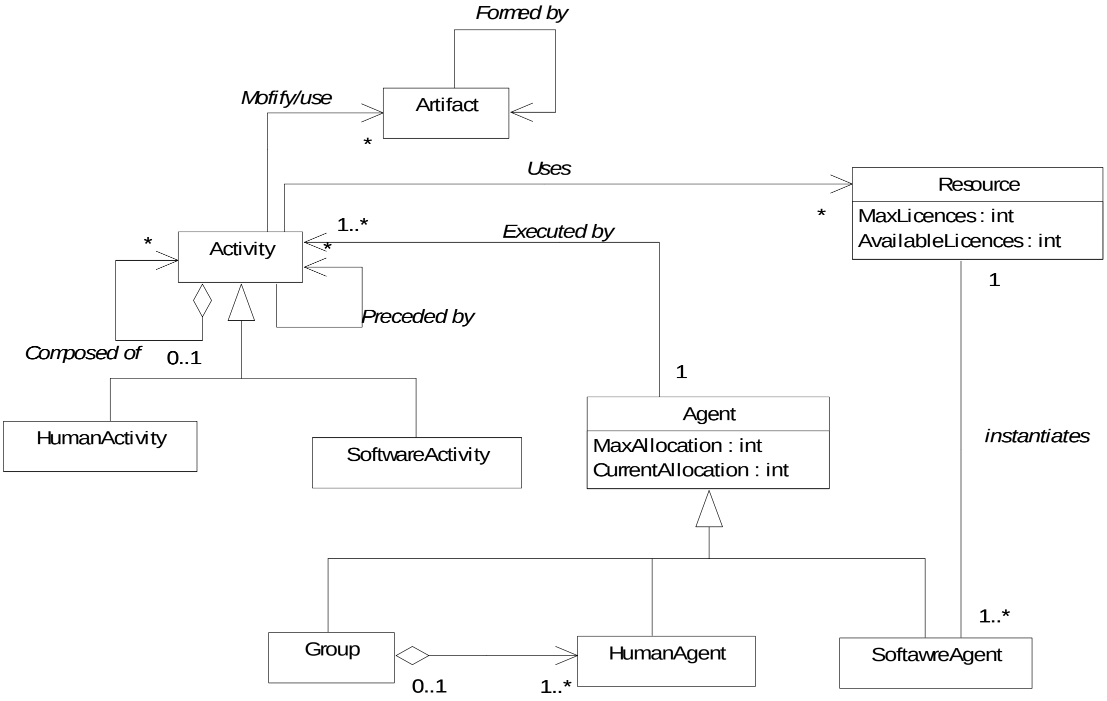


Figure 4.6. OPSS predefined classes representing basic process constituents

Process Elements

Process elements in this approach are provided in terms of UML classes through a predefined class diagram (cf. figure 4.6). These classes can be extended by means of the UML specialization/generalization relationship in order to add new types of process elements (i.e., specific roles, activities, etc.). However, there is not a proper semantic for these elements and no adapted notation. They all have the same semantic as the UML *Class* metaclass since they are all instances of this metaclass. Almost basic process elements are given and are represented as a UML class. The notion of *Software Agent* means a ***Tool***. The notion of ***Role*** that process agents may undertake is lacking.

Activities and Actions Coordination

Regarding the proactive control, the *finish-start* precedence relationship is provided through the sequence control flow used in the UML activity diagram. Semantics of the UML *Join* and *Fork* elements is enriched to describe the possibility of having multiple instances of the same or different activity type that are enabled for execution in parallel. This is to model the *start-start* precedence relationships between activities. In state diagrams, events are used as a means to trigger transitions allowing activity state changes. However, there is no link between state changes of distinct activities, which limits the reactive control of this approach.

Exception Handling

Exception handling is not addressed by the approach. However, one can imagine a state called "exception", which once reached can have as an action, the call of an operation to warn that an exception occurs. In Di Nitto's approach, the set of states proposed for each entity is fixed and cannot be changed.

Advanced Constructs

Expressing conditions is made possible thanks to UML Activity diagram decision nodes. However, notions such as loops, iterations, tool invocations are not provided by the language.

3.3.1.2. Understandability

Modeling the process constituents by specializing a predefined UML class diagram opens the way to many UML-familiar people to model their processes easily. The activity diagram is used to describe the flow of work within the process. However, there is not link between the two diagrams. Authors claim that the link between activities defined in the class diagram and those defined in the class diagram is checked through name matching. It may work for one simple class diagram, but in case of combining many class diagrams to form one global process definition this may turn out very difficult.

Additionally, authors claim that using the composition relationship in the class diagram allows defining activities that can be composed by many other activities, each activity having its own performer (agent). However, this composition aspect cannot be reflected in the activity diagram. In the activity diagram, one activity, may it be simple or composed, is realized by one and only one role materialized by the swim lane. Thus, representing a compound activity with each of its component activities having a different performer is rendered impossible by the activity diagram. Here again, the relationship between the two diagrams is weak.

Finally, the use of state diagrams for modeling the process business rules may penalize a bit the ease-of-use of this approach. Indeed, state diagrams are more complicated to define than activity and class diagrams and not all people are familiar with them. The states are fixed, but process modellers have to define events, guards, actions of each state in a very precise manner since that the process execution depends on it.

3.3.1.3. Precision

The unique unit of work is the activity. It may represent the simple atomic action as well as activity composed of many other activities. However as said previously this can only be expressed in the class diagram. The state diagram provides a very precise means to describe activity's states and events that may trigger transitions for state changes. In the class diagram, classes can be specialized, new attributes and operations can be added in order to adapt the class diagram to your specific process preoccupations.

3.3.1.4. Executability

To execute the process, all the diagrams are used for generating the code. User-defined classes derived from predefined classes to describe specific elements of the process being modeled are translated into the corresponding Java classes, equipped with attributes, methods and associations as described in the given UML class diagrams. The body of new methods is defined according to the information provided by the corresponding state and activity diagram. However, the black point in this translation is how precedence relationships (sequencing of activities) defined in the activity diagram are reported in the Java code. Unfortunately, the only reference to this by the authors is: *"relations represented in the activity diagrams are translated into java code which manages such relations"* [Di Nitto 02].

Another lack of this approach in terms of executability is that the code of new operations introduced in user-defined classes can only be inferred from the state diagram of the class. If the modeller does not provide enough information in the state diagram, the implementation of the operation is left incomplete: the process modeler is supposed to complete this implementation after the translation has been performed

3.3.1.5. Modularization

Modularization and process compositions were not explicitly addressed in the author's work. We can figure that to combine different processes, one can simply make all the process classes from the different processes specializing the predefined OPSS classes (i.e., *Activity, Artifact, Agent, etc.) w*ithin the same class diagram. However, two points have to be taken into account. The first one is to make sure that two specialized process constituent classes have not the same name. The second is to establish the precedence relationships between all the activities defined in the class diagram within the activity diagram. The last point cannot be automated.

3.3.2. Discussion

The advantage of this approach is that process constituents can be defined easily by simply specializing a set of predefined classes provided by the approach in form a UML class diagram. The flow of work is given in activity diagrams and the lifecycle of each entity is defined by a state diagram. However, we have demonstrated through an evaluation that the activity and class diagram have no links with each other.

The approach does not extend the UML language nor introduces new concepts. Process elements are simply instances of the UML *Class* metaclass, which means that they all have the same semantics and notation as the UML *Class* metaclass.

Regarding execution, it is essentially based on how state diagrams defined by the user are precise enough and sound in order to enable a complete code generation and to allow process execution within OPSS. Otherwise, code has to be added manually. The black point in the executability aspect remains how information defined in activity diagrams (i.e., precedence between activities), state machines and class diagrams are integrated to generate each of the Java classes needed for the execution. Authors did not precise how this integration is realised. Modularization is not addressed by the approach.

3.4. Promenade Approach

Promenade stands for (**PRO**cess-oriented **M**odelling and **EN**actment of software **DE**velopment). It is a software process modeling language defined in the context of the ComProLab project [Franch 97, Franch 98]. Promenade consists of two different parts: the static one, which introduces main elements that play a part in the process (i.e., *Agent*, *Task*, *Document*, etc.); and the dynamic one, which establishes the order in which tasks are to be enacted. The static part comes in form of UML1.3 class diagram [OMG 00b], which comprises a set of predefined classes. These classes can be specialised in order to meet specific process requirements. The dynamic part defines what the authors call a precedence graph, which defines the different kind of precedence relationships that may exist between process's tasks. To some extents, the static part of this approach is very similar with Di Nitto's one. Thus, the same criticisms we developed in the previous approach are also valid for the static part of Promenade. In the following, we evaluate this approach according to SPML requirements.

3.4.1. Promenade Evaluation

As we said earlier, this approach is very similar to the one presented by Di Nitto's team. Thus, when needed, we will refer to the evaluation made in the previous subsection (cf. section 3.3.1).

3.4.1.1. Semantic Richness

The static part of the language is built upon three kinds of information, which yield to several complementary UML class diagrams. First, individual information of predefined classes is given (i.e., classes with there respective attributes and operations) including constraints (e.g., invariants). Second, a UML class diagram is defined for describing the hierarchy (generalisation /specialisation) between the different constituents of the process (figure 4.7). New classes inheriting these predefined classes can be added for specific process needs (i.e., specific roles, artifacts, etc.). Finally, a class diagram is used for describing association relationships between predefined classes (figure 4.8). This diagram can also be extended.

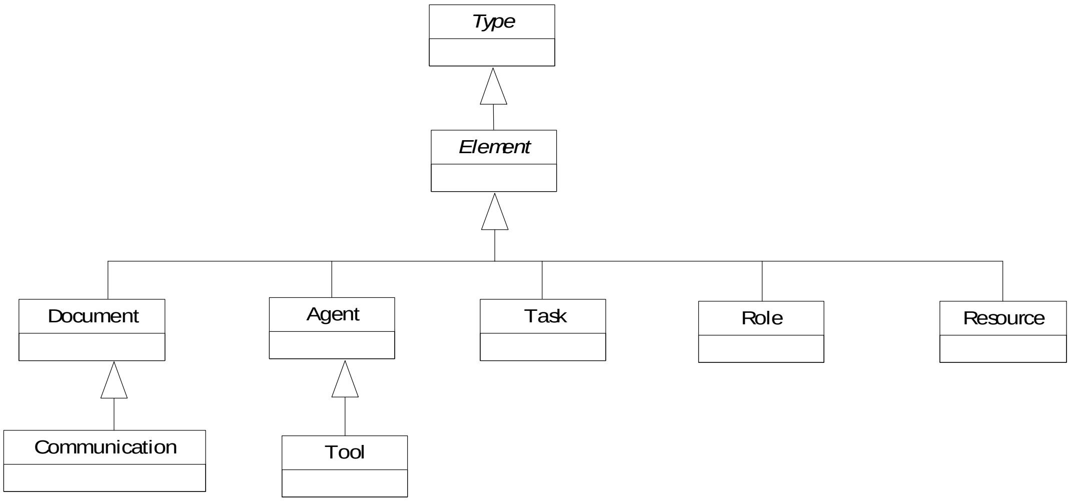


Figure 4.7. Default UML Class Hierarchy Diagram of Promenade predefined classes

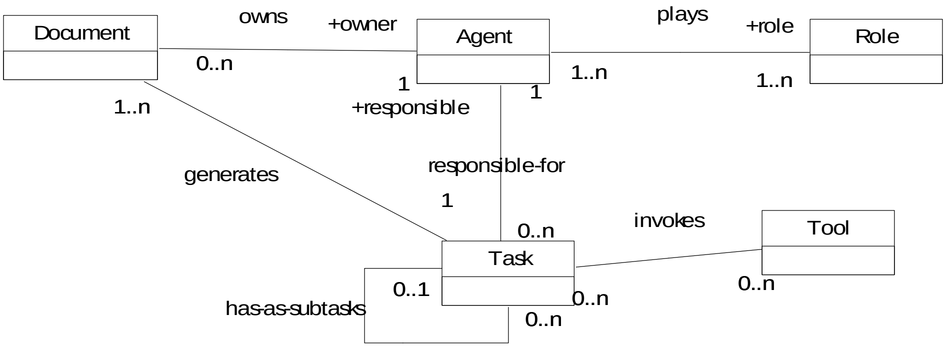


Figure 4.8. Default associations between Promenade predefined classes

Process Elements

As in Di Nitto's approach, process elements in Promenade are given in terms of instances of the UML *Class* metaclass. Thus, no proper semantics (distinct from the

UML *Class* metaclass semantics) or notation can be assigned to these concepts. Here the notion of *Document* represents the notion of ***Artifact***. All process elements are provided through the predefined UML classes introduced by the language (figure 4.8).

Activities and Actions Coordination

The dynamic part of the language is in charge of establishing the order in which process tasks can be enacted. *Precedence* is the means for representing proactive control specification in Promenade (c.f., Figure 4.9.). The approach defines four kinds of precedence. The *Strong* precedence for defining the basic *finish-start* precedence between tasks. The *Weak* precedence to model the fact that task *S* must be started before *T* and must end before *T*. The *Synchronizing* precedence, which distinguishes two types of precedence, the *start-start* and the *finish-finish* precedence relationships. The *start-finish* precedence is not represented.

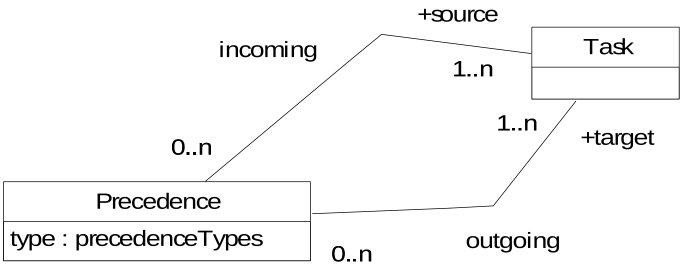


Figure 4.9. Precedence between process tasks

The approach proposes another means to ensure a proactive control. It consists of including two attributes in each task. The first one to store all input documents of the task and the second one to store all outputs of the task. Then, authors proposes to link tasks according to their inputs/outputs (if task A has an output which is the same as the input of task B then task A is linked with task B). Definitely, this solution can cause many disagreements. The first one is because a task having its inputs equivalent to another task's outputs does not necessarily mean that a direct precedence relationship exists between them. Second, we can have many tasks having as input the same output of a task. Thus, which one to choose? Third, in case of combining many processes, process modellers have to rename process's documents, which may represent the same document but are named differently from one process to another, in order to ensure the precedence relationship between tasks. Thus, basing precedence relationships between tasks on document names is not a reliable solution.

Reactive control is ensured by means of activity's state diagram. However, authors do not give information about events and actions used for defining state changes.

Exception Handling

Promenade defines a predefined state diagram, which represents the set of allowable task's states. One of these states is the *"CompleteUnsucc"* state, which once reached, may cause the call to an operation that can handle the unsuccessful completion of the task.

Advanced Constructs

No tool invocation mechanisms, no loop constructs or conditionals (e.g., switch like element) are proposed by the approach.

3.4.1.2. Understandability

The approach is quite simple and supposes that the process modeller is familiar with UML class diagrams. To model a process, one has just to specialize the set of predefined classes provided by the approach. To define precedence between process's tasks, one has to define a precedence graph, which defines the order between all tasks of the process. Precedence rules are described using a declarative formalism, which is quite simple. However, authors do not specify how the precedence graph (including precedence rules) is to be integrated with the class diagram to form a complete process model. In our view, using a class diagram for describing the flow of work within the process is not a very elegant and a precise way. Additionally, the process modeller has to combine between the two diagrams (i.e., class and precedence graph) in order to understand and to reason about the process, most of all that there is no link between these diagrams.

3.4.1.3. Precision

The unique unit of work is the *Task* class, which is an instance of the UML *Class* metaclass. A *Task* can be composed by other tasks however; the precedence between these tasks is to be defined separately in the precedence graph.

3.4.1.4. Executability

Executability was not addressed at all by the Promenade approach. No prototype was implemented and no tool is provided.

3.4.1.5. Modularization

For combining many processes, authors propose to make all process elements of the same type from the various processes, inheriting the same predefined classes defined by Promenade. Thus, the static part of P (P: the new combined process) $P=P_1+P_2+\ldots+P_n$ is the superposition of their generalisation hierarchies, together with the union of their association and aggregation relationships. However, to avoid name clashes (of classes, relationships, attributes, etc); authors propose to rename (to give a similar name) all the classes that represent the same artifacts but that are called differently from one process into another. This requires manual intervention to identify these classes and to rename them. What makes the thing more complex is that the precedence graph has also to be modified separately in order to ensure that for instance the document output of task A from process P_2 (which was renamed) is the input document of task B from process P_3. This definitely may revel to be a very tedious and unmanageable task.

3.4.2. Discussion

This approach presents many lacks. The first one comes from the fact that the approach does not provide a proper set of concepts for describing process elements but relies on a UML class diagram with a predefined set of classes (that can be extended). The second lack is that the approach does not provide any mechanism or way to execute Promenade process models. Modularization revels to be very complicated and cannot be automated. Understandability may be affected since process modellers have to deal with a set of class diagrams and precedence graph in order to understand the process's flow of work. Finally, no tool or prototype is provided.

3.5. Shih-Chien Chou's Approach

Chou's work was realised in the context of a research project financed by the National Science Council of Taiwan [Chou 02]. It proposes a software process modeling language consisting of high-level UML-Based diagrams and a low-level process language. While UML diagrams are used for process's participants understanding, the process language is used to represent the process - from UML diagrams – in a machine-readable format i.e., a program.

3.5.1. Chou's Approach Evaluation

As it was done for the previous approaches, we evaluate this approach according to the following criteria.

3.5.1.1.Semantic Richness

For the high-level part of the language, the author proposes the using of two diagrams called *P-activity* diagram and *P-class* diagram. These diagrams are respectively based on a subset of the UML1.4 *Activity* and *Class* diagrams [Chou 00]. The P-activity diagram is used to model activities, their sequencing and synchronization, events and exception handlers. The P-class diagram is used to model products, roles, tools, schedules, budgets and their relationships. All these elements are in fact represented as UML classes with attributes and operations. Thus, in a P-class diagram, you may have a class named "Analyst" which is linked with a named and directed association (e.g., responsibleFor) with a product called "Specification", which in its turn is represented as UML class with its attributes (e.g., specName, projectName, etc) and operations (e.g., editSpec(), createSpec(), etc).

At the end, modeling you process's constituents (i.e., roles, workproducts, etc) will be viewed the same as if you were modeling any application's structure using a class diagram in a traditional design phase. The P-activity diagram will determine the flow of work, i.e., sequencing of activities using an UML Activity diagram. However, there is no link between the two diagrams (i.e., P-class diagram and P-activity diagram)

The set of UML elements and their notations used for defining both P-x diagrams are represented in figures 4.10 and 4.11.

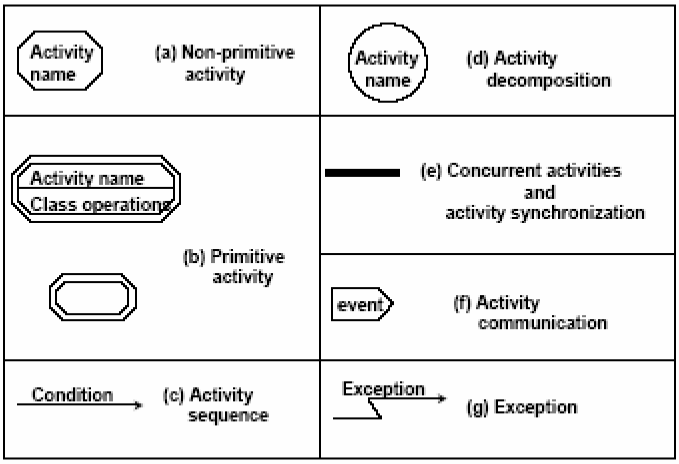


Figure 4.10. P-activity diagram notation

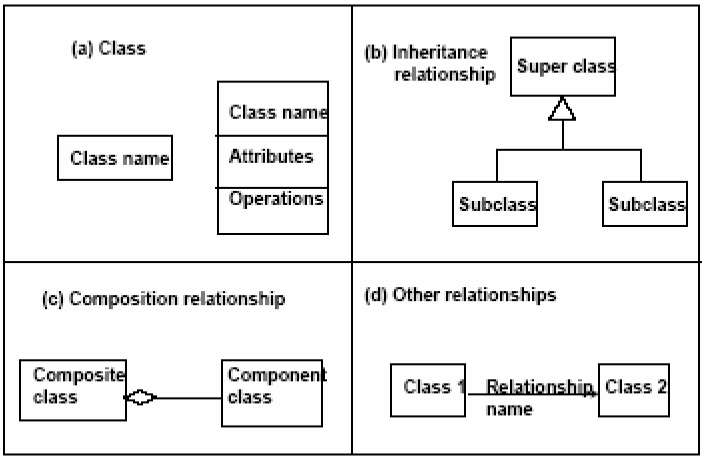


Figure 4.11. P-class diagram notation

At the lower level, the author uses a proprietary and minimal object-oriented language for representing the process as a program. The language is described in terms of BNF (Backus-Naur Form) grammars. A process program represented in the language is composed of one `Process` class and one or more other classes which can be `Role` classes or `Product` classes. To represent the synchronization between process's activities and their sequencing in a `Process` class, a `Task` is used. A `Task` is defined as an operation of the `Process` class. A `Task` may be a concurrent block (i.e., for calling several process's activities concurrently), an event statement (i.e., a wait or a sending of a signal), etc. Finally, all process's activities are defined in terms of operations invoked form `Task` blocks within the `Process` class.

Process Elements

At the higher level, the language proposes the concept of *Activity* through the P-activity diagram. The *Activity* element has no additional attributes than those defined in the UML1.4 Activity concept [OMG 01]. The notions of *Role* and *Document, Tool* are in fact UML classes described in a class diagram called P-class diagram. These classes can then be extended for a specific project in order to add specific roles (e.g., *Analyst, Designer,* etc.), products (e.g., *Specification*, *Requirement*, etc.), etc as well as relationships between these classes (e.g., *isResponsibleFor*, *isEditedWith*, etc). This is very similar to Di Nitto's approach (c.f., Section 3.3.). Thus, these concepts (i.e., *Role, Document,* etc.) have no appropriate metaclasses representing them, but are only instances of the UML1.4 *Class* metaclass. The notion of ***Agent*** is lacking.

Activities and Actions Coordination

In the P-activity diagram, proactive control is ensured thanks to *Activity sequence* (cf., figure 4.10 (c)) and the *synchronization* and *concurrent bar* elements (cf., figure 4.10 (e)). The former allows the traditional *finish-start* precedence relationship between activities. The latter can be used to describe a *start-start* (concurrency) precedence relationship between activities. The reactive control is ensured using events and exception handlers. Events can be used to model the *finish-finish* and *start-finish* dependencies between process's activities.

At the lower level, the *Activity sequence* is expressed as an operation call to the process's activity (represented by a process class operation). Concurrency is expressed in terms of a specific kind of `Task` defined by the `concurrent {}` block. This block will then calls the concurrent activities represented as operations within the process class. Other kinds of `Tasks` are used to express event statements and exception handlers. Details can be found in [Chou 02].

Exception Handling

Exception Handlers are represented as activities in P-activity diagrams. In the process program, they are expressed as operations. They are triggered explicitly from operations in case of an unexpected result (e.g. validation of the design fails).

Advanced Constructs

In P-activity diagrams, there is no means to express loops. Conditions are expressed as strings on top of the *Activity sequence* arrow (cf., figure 4.10 (c)). In the BNF language's grammars, the `Branch` and `Loop` statement are defined. However, the author does not refer to them within the paper and no example is provided.

3.5.1.2.Understandability

The language does not extend UML to define a new language for software process modeling. Rather, it proposes the use of UML Activity and Class diagrams. The former is used for modeling the flow of work and the latter for modeling the different constituents of the process as instances of the UML *Class* metaclass. These diagrams represent the high-level part of the language and are very simple to understand by process participants whom are familiar with UML.

However, there is no link between the two diagrams. They are just used for the process comprehension. Moreover, there is no automatic generation of process programs from these diagrams towards the proprietary OO language the author proposes. This imposes that process modellers need to be familiar with both languages (UML and the process language) since they have to rewrite the process using the low-level process language. Besides, every time you add a new class (i.e., a new role, product, tool, etc.) in the P-class diagram, you have to code its equivalent class in the process language and to make sure that it is properly linked with other process constituents (i.e., roles, products, activities, etc.). This requires mastering the process language, which may be an obstacle for the adoption of the approach by organizations.

3.5.1.3. Precision

In this approach, the activity element is the unique concept used for describing the process hierarchy at the higher level (i.e. graphical notations using P-activity diagram). A process can be first depicted as a top-level P-activity diagram, which is composed of coarse-grained activities (i.e., non-primitive activities using notation in figure 4.10 (a)). Activities in the top-level diagram can be decomposed to form more detailed P-activity diagrams if necessary. The decomposition proceeds until all activities are fine-grained enough (i.e., primitive activities using notation in figure 4.10 (b)).

However, at this level, it is not possible to know neither what are the products (artifacts), inputs/outputs of each activity nor the roles in charge of each activity. In our view, these aspects of the process need to be highlighted in process models in order to increase the agent understanding about the process and to ease the reasoning about the eventual process issues.

At the lower level, roles and products are defined as classes and are instantiated within the process class. However, products are not used as parameters of process's activities (represented as operations within the process class). Instead, they are used as Role's operation parameters (e.g., a role's operation call: *Analyst1.EditSpec (systemReq,* *subSpec1))*. This imposes that activity steps are implemented as operations within roles in charge of the activity. This hardly limits the reusability of role classes since they have to be surcharged with operations specific to the activity carried out by the role.

3.5.1.4. Executability

While the P-class and P-activity diagrams are provided as a means to reason about the process, the approach does not provide an automatic generation of the process program form these diagrams. The process program has to be implemented by developers according to what is defined within the diagrams. There is no means to reflect changes or additions made to the process program into P-activity and P-class diagrams. The author does not talk about any tool or prototype implementing the approach.

3.5.1.5. Modularization

Modularization is not addressed by the author. We suppose that it is possible to combine process programs at the lower level of the language since it is an OO-based language. Once the process program we want to integrate is instantiated, its operations representing the implementation of the process's activities can be invoked.

3.5.2. Discussion

Whether the language provides the advantage of directly using UML diagrams for modeling the process aspects (i.e., flow of work and constituents), the approach presents some lacks. The first one is the fact that process constituents are represented as instances of the UML *Class* metaclass which may not fit the semantics of software process constituents. Besides, these concepts are not part of the language since they are instances of the UML *Class* metaclass. The second obstacle of this approach is the lack of an automatic generation of process programs from P-x diagrams, which imposes the rewriting of the process by developers mastering the proprietary OO language the author proposes. Any addition to the P-class diagram imposes the coding of a new class and most of all, its linking with the other process classes. This can turn out very complex in case of large P-class diagrams or in case of constant changes of information within these diagrams. Finally, no prototype is provided and no further works were proposed for this approach.

4. Discussion

In the previous section, we evaluated UML-Based approaches for modeling and executing software processes. Each approach presents some advantages but also some drawbacks. Thus, we find it interesting to highlight some observations we had. The first observation is that three of the five approaches we discussed (i.e., Di Nitto et al., Promenade and Chou approaches) do not define new concepts or extend UML metamodel ones. They simply consist in providing a predefined UML class diagram for defining principal process constituents (i.e., *Activity, Role, Artifact, Agent or Tool*) in terms of instances of the UML1.x *Class* metaclass (i.e., simple UML classes). Thus, these process elements do no have a proper semantics and notations, which are actually borrowed from the UML *Class* metaclass. The second observation is, for these three approaches, there is no link between the different UML Diagrams they use. Some of them propose name matching from one diagram into another, which of course cannot be a reliable solution. In the case of Promenade for instance, the process modeller has to combine between a class diagram and *precedence* graph described thanks to precedence rules in order to get a view of the flow of work within the process.

Almost approaches propose all process elements except the notion of *Agent*, which lacking in three approaches (SPEM1.1, SPEM2.0 and Chou's approaches) among the five. *Exception handling* is also missing by all the approaches except for Chou's one. *Proactive control* is more addressed than *Reactive contro*l, which in most cases (for the approaches they provide it) is ensured by means of events modelled within state diagrams. Most SPMLs do not provide constructs for modeling loops, conditionals, tool invocations, agent communications, etc.

Regarding *understandability*, SPEM1.1 is quite a good compromise. It offers simple concepts through a MOF metamodel extending some UML1.4 concepts. However, some of its element semantics present some ambiguities [Bendraou 05]. The lack of SPEM1.1 is that it does not offer any process model execution approaches and its approach for composing process models presents some obstacles. Conversely, SPEM2.0 provides panoply of mechanisms for extending and combining chunks of process and method descriptions however, this adds to the complexity of the standard. Indeed, the standard proposes three compliance points, seven packages and many metaclasses, which makes it unreadable. Additionally, SPEM2.0 does not reuse UML Superstructure concepts, which provide through the *Activity* and *Actions* packages all the necessary concepts for dealing with process's activities sequencing, constraint expressions, events, and so on. Moreover, SPEM2.0 is not executable. We demonstrated the limits of the approaches proposed by the standard for SPEM2.0 process model executions. Regarding executability in the other approaches, we saw that Chou's approach consists in rewriting manually the process program from the UML diagrams, which is inconceivable. Di Nitto approach consists in generating code from the three UML1.3 diagrams used for describing the process (i.e., *Activity, Class* and *State Machine*). However, no information is given about, how process aspects (i.e., activity sequencing, events, actions, class's operations and attributes) defined in these diagrams are translated and integrated into the Java code. The Promenade approach does not provide any execution possibilities.

Looking at the *Abstraction*, SPEM1.1 and SPEM2.0 are the only SPMLs that provide a set of concepts with their own semantics and notations through a metamodel instead of simply using UML diagrams, which, are at the model level (i.e., process concepts are instances of UML Class metaclass). Unfortunately, this raise in abstraction was not followed by an automatic generation of code or formalism from SPEM process model execution. As we addressed in the previous chapter, whether the MDE claims for raising the abstraction level of modeling languages, it also insists on the necessity of keeping a tight relationship with lower level languages to ensure model executions

Finally, regarding the *Tooling Support*, only the industrial standards (i.e., SPEM1.1 and SPEM2.0) have some implementing tools. However, these tools are more often used for drawing process models (contemplative models) without any execution support behind. Additionally, the standard implementation differs from one tool into another and process models are stored in proprietary formats rendering process model exchanges impossible, which is in opposition with MDE principles.

All the discussion and evaluation we made is summarized in table 4.1.

Approaches	**SPEM1.1**	**SPEM2.0**	**Di Nitto's et al. Approach**	**Promenade**	**Chou's Approach**
Requirements					
Semantic Richness					
Process Elements		Depends on the Compliance point used	provided in terms of UML1.3 classes (Instances of the UML *Class* metaclass)	provided in terms of UML1.3 classes (Instances of the UML *Class* metaclass)	provided in terms of UML1.4 classes (Instances of the UML *Class* metaclass)
Activity	Work Definition, Activity. An Activity may be composed of *Steps*	Activity / Task Definition	Activity	Task	Activity
Role	Process Role	Role Use / Role Definition	**No**	Role	
Artifact	WorkProduct	WorkProduct Uses / WorkProduct Definition	Artifact	Document	Document
Agent	**No**	**No**	Human Agent	Agent	**No**
Tool	**No**	Tool Definition	Software Agent	Tool	Tool
Activities/Actions Coordination					
Proactive Control	*Precedes* dependency. Kinds of precedence ensured: *start-start, finish-start* or *finish-finish*. The *start-finish* precedence is lacking	Ensured thanks to the *WorkSequence* concept. Kinds of precedence: *start-start, finish-start* or *finish-finish* and *start-finish*	Use of UML1.3 Activity diagram sequence control flow for modeling *finish-start* precedence. UML *Join* and *Fork* for modeling *start-start*	Use of the *Precedence* concept. Allows modeling the *start-start, finish-start* or *finish-finish*. the *start-finish* is lacking	Ensured thanks to UML *Activity sequence*¸ fork and join elements for modeling *start-start* and *finish-start.*
Reactive Control	**No**	**No**	In state diagrams, events are used as means to trigger transitions allowing activity state changes	An predefined state diagram is defined for Task. However, no information is given about event, action, and states changes	Through events and exception handlers for modeling *start-finish, finish-finish*
Exception Handling	**Not addressed**	**Not addressed**	**Not addressed**	**Not addressed**	Exception Handlers are represented as activities in AD and as operations in the code
Advanced Constructs (Tool Invocations, Agent Communication, Loops, Conditionals)	No advanced constructs provided	No advanced constructs proposed. However the *ExtensibleElement* metaclass provide the possibility to add user-defined concepts and process specific constituents (i.e., specific roles, workproducts, etc)	Decision nodes through Activity Diagrams Notions such as loops, iterations, tool invocations are not provided by the language	No advanced constructs provided	Loops and Conditionals are proposed in the OO proprietary language the author proposes for process execution.

Understandability	**Good.** Simple Metamodel, reuse of UML1.4 diagrams for process descriptions. Some ambiguities about some concepts (e.g. *Process Performer*) identified in [Bendraou 05] and [Combemale 06]	**Lack:** Very complex and hard to tackle. One has to deal with many compliance points. The specification defines seven packages with seventy-five metaclasses. The limit between process description and method definition is confusing	**Good**. Use of UML diagrams (*Activity*, *Class* and *State machine* diagrams) for modeling process aspects. **Lack:** no link between the diagrams. The reliability of the code generated for process execution depends on how the process modeller defines state machines	**Lack:** Process modeller has to combine between a class diagrams and precedence graph in order to understand and to reason about the process. No link between the two diagrams	**Good**. Use of UML Activity and Class diagram for modeling the process. Use of proprietary OO process language for process execution. **Lack**: no link between the diagrams. No automatic code generation form diagrams.
Precision	Process hierarchy described through *Phase*, *Lifecycle*, *Iteration*, and *WorkDefinition* to *Activity*, and *Step* concepts. **Lack:** There is no way to express what are the inputs/outputs of Activity's steps, who is the role responsible of each step neither their constraints	**Good.** Using "*SPEM Process with Behavior and Content"* compliance point, the *Activity* is the only concept for describing the hierarchy of the process model. Using the *"SPEM Method Content"* or *"SPEM Complete"* compliance points it is possible to describe the *Method Library* in terms of *Tasks* that it may contain *Steps*	**Good**. The unique unit of work is the *Activity*. The state diagram provides a very precise means to describe activity's states and events that may trigger transitions for state changes. In the class diagram, classes can be specialized, new attributes and operations can be added	The unique unit of work is the *Task* class	At the higher level, Activity as unique unit of work. At the lower level, roles and products are defined as classes and are instantiated within the process class. Activities implemented as process's operations
Executability	**Lack:** Out of scope of SPEM1.1	**Lack:** Not addressed. Proposition of two approaches. We demonstrated their limits (c.f., section 3.2.1.4.)	Code for process execution is generated from Activity, Class and State machine diagrams. **Lack:** no details on how the code is generated and how different information from the different diagrams are integrated to the Java code. If the state diagram is not complete, code has to be added manually after generation	**Lack:** Not addressed	**Lack:** the approach does not provide an automatic generation of the process program from these diagrams. Process has to rewrite the process using a proprietary OO language
Modularization	Through the notion of *Process Component (PC)*. **Lack:** to combine Process Components a *Unification* phase is needed (i.e., renaming all *Process*	**Good**. Depends on the compliance point used. For concepts for extending process activities, one has to use the *Activity*'s *"Activity*	**Not addressed**	**Lack:** Based on renaming classes from different class diagrams to be combined for defining a new process	**Not addressed**

	Roles and *WorkProducts* to have identical names, linking explicitly activities from different PCs)	*Use Kind"* property (enumeration). To extend, reuse or replace process contents, method libraries, one has to employ *Variability* mechanisms. For process composition, one can use *Process Component* and *Method Plugin* mechanisms			
Metamodel/ Profile	A MOF-Compliant metamodel extending UML 1.4 concepts / A UML1.4 Profile is proposed	A MOF-Compliant metamodel extending the UML2.0 Infrastructure (no concept reused from Superstructure) / A UML2.0 Superstructure Profile **Lack:** OCL constraints are not provided (up to standard implementers)	**Lack:** No metamodel, no profile provided. Authors provide a set of predefined classes (instances of the UML *Class* metaclass) that one can specialise for a specific process	**Lack:** No metamodel, no profile provided. Authors provide a set of predefined classes (instances of the UML *Class* metaclass) that one can specialise for a specific process	**Lack:** No metamodel, no profile provided. Author provides a set of predefined classes (instances of the UML *Class* metaclass) that one can specialise for a specific process
Abstraction	**Good.** Definition of a new language using a Metamodel with a proper semantics and notations. Focusing only on *Software Process Modeling* aspects	**Good.** Definition of a new language using a Metamodel with a proper semantics and notations. Focusing only on *Software Process Modeling* aspects	**Lack:** concepts have neither proper semantics nor notations. All concepts are instances of the UML *Class* metaclasses then, all have the same semantics and notations.	**Lack:** concepts have neither proper semantics nor notations. All concepts are instances of the UML *Class* metaclasses then, all have the same semantics and notations. It is up to the modeller to write precedence rules in a proprietary formalism.	**Lack:** concepts have neither proper semantics nor notations. All concepts are instances of the UML *Class* metaclasses. Process modeller has to manually write the process program
Is a Standard/ Standard-Based	Is a standard / Based on the UML1.4 standard	Is a standard / Based on the UML2.0 Infrastructure for the metamodel, based on the UML2.0 Superstructure for the Profile	Uses UML1.3 standard diagrams	Uses UML1.3 standard diagrams	Uses UML1.4 standard diagrams
Tooling Support	Rational Process Workbench, IRIS Suite, Objecteering's SPEM Profile	Eclipse Process Framework, Objecteering's PRO3	ORCHESTRA Process Support System for process execution after Java code generation. No translator (from UML diagrams to Java code) is provided	No tool or prototype is provided	No prototype is provided

Table 4.1. Evaluation and Comparison of UML-Based approaches for Software Process Modeling and Execution

5. Conclusion

The first chapter of this document aimed at presenting the different process technologies. We saw the different characteristics of each of them and most of all; we clarified the relationships between a *Business Process*, a *Software Process* and a *Workflow*. We concluded that each process technology has its specific preoccupations and concerns, which justifies that many process modeling languages are proposed for each of the three domains. Comparing between PMLs from the different process technology domains would not be objective. We fixed our domain, which is *Software Process Modeling*.

In the second chapter, we introduced MDE and we presented its principles. If respected, these principles can considerably improve productivity and decrease complexity in developing software. We highlighted how the software process modeling community was attracted by such promises and how it can take advantage of the MDE vision. One of the MDE principles is the use of high-level and standard formalisms for modeling purposes. In the last decade, the UML succeeded to be that reference high-level modeling language. This has naturally influenced the SPM community to explore the possibility to reuse UML for process modeling and many approaches were proposed.

In this chapter, we compared and we evaluated the UML-Based approaches for software process modeling and execution. These approaches were compared using a framework we defined. This framework regroups major requirements to fulfil while designing a process modeling language that we collected from well-known works in the literature. These requirements are combined with MDE principles introduced in chapter 3. During the evaluation, we did not limit ourselves to simply describing the approaches but we identified advantages and drawbacks of each of them. Finally, we concluded that no approach succeeds in fulfilling the requirements we defined. More particularly, no approach succeeds in offering a high level of abstraction in modeling software processes while providing means to execute process models, which is a key principle within the MDE approach. Thus, a kind of trade-off is needed between *Abstraction* and *Executability*.

In the remaining part of this document, we present our UML-Based proposition for Software Process Modeling and Execution.

.

Chapter 5

UML4SPM, a UML2.0-Based Language For Software Process Modeling

1. Introduction

After the introduction of process technology domains and the presentation of the different UML-based approaches for software process modeling, in this part of the document, we present our proposition.

UML4SPM is the UML2.0-Based language we defined for Software Processes Modeling and Execution. It comes in form of MOF-compliant metamodel extending the UML2.0 Superstructure standard. In this chapter, we start the presentation of our solution by giving main motivations that led us to propose UML4SPM. These motivations are then fixed as design goals and are introduced in Section 2. In Section 3, the presentation of UML4SPM metamodel is given in two parts. The first part introduces the UML4SPM MOF-compliant metamodel and describes in details each of its metaclasses. The second part presents UML2.0 Superstructure concepts we extended and we reused in the UML4SPM definition. In Section 4, we introduce the UML4SPM notation, which is principally inspired from the UML2.0 *Activity* notation and which we enrich with some features proper to software process modeling. Section 5 concludes this chapter.

2. UML4SPM: Design Goals

The initiative of developing UML4SPM emerged after many observations we had while exploring the different software process modeling languages proposed by the literature. These observations were also confirmed by our industrial partners from the European projects we have been involved in and in which, UML4SPM was part of [Modelware, Modelplex]. While developing UML4SPM, these observations became our main design goals. We present them in the following:

- The first one was that first-generation SPMLs were too complicated to be understood and to be used by non-experts in computer science. They were based on low-level formalisms and required many programming skills. This observation sets our first requirement which is the need of raising the ***Abstraction*** level of process modeling languages in order to increase *Understandability*.

- The second observation was that using a ***Standard*** and ***Well-Known*** formalism would make the PML adoption easier and at lower costs. People do not have to learn a new language and leveraging existing tools is rendered possible. In UML4SPM, we opted not to start from scratch. Thus, we investigated the possibility of reusing a standard, powerful and already very popular modeling language, which is UML2.0 [OMG 07a, OMG 07b]. In [Bendraou 05], we demonstrated that the newly adopted standard has a high potential as a basis for a

SPML through its *Activity* and *Action* packages, which have radically changed from UML1.x previous versions. In UML2.0, *Activities* are inspired from *Petri Nets* and are not only used to model processes, now, they also have some features necessary to support the automation of these processes [Störrle 04].

Another reason that makes us reusing UML2.0 was our aim to participate in the OMG's revision of SPEM1.1 (Software Process Engineering Metamodel) standard [OMG 05a], namely, SPEM2.0. The SPEM2.0 RFP (Request For Proposal) imposed as mandatory requirements - among them - defining a MOF-compliant metamodel for introducing concepts proper to software process modeling and where needed, reusing of UML2.0 *Activities* in defining the SPEM2.0 metamodel. The RFP requirement stated: *"submissions shall rework activity definition and modeling -from SPEM1.1-, so as to take advantage of the new UML 2.0 Activity Diagram features"* [OMG 04]. We have considered these requirements in defining the UML4SPM metamodel.

UML4SPM takes advantage of the expressiveness of UML2.0 by extending a subset of its elements suitable for process modelling (i.e., *Activities* and *Actions*). By adopting UML2.0 as a basis of our SPML, we take advantage of:

- ➢ The expressiveness of the UML2.0 in modelling sophisticated activities including actions with executable semantics and in orchestrating them;
- ➢ The fact that UML is currently the most widely used modeling language in the industry. People are familiar with the language and a myriad of tools and training supports are provided;
- ➢ Notations and diagrams offered by the standard. UML diagrams are intuitive and easy to understand;
- ➢ Easier adoption by UML and SPEM1.1modelers;

- ▪ Finally, the last observation we have noticed is the increasing demand from industrials for *executable* process models instead of *contemplative* process models. This demand is motivated by the continuous requests for more complex yet reliable software in short time-to-market. Nowadays, companies are looking for how to extensively automate all parts participating in the software production, including the development process itself. However, ***Executability*** must not alter the *Simplicity* and *Understandability* of the PML resulting from satisfying our first design goal, which is raising the *Abstraction* level of the PML. Thus, a trade-off between *Executability* and *Abstraction* is needed.

The combination of the three design goals we just introduced i.e., *Abstraction, the use of Standard formalisms* and *Executability* is key in applying the MDE vision for the Software Process Modeling domain. Advantages of MDE and its principles were introduced in Chapter 3.

In the following, we present the UML4SPM metamodel.

3. UML4SPM: The Metamodel

The UML4SPM metamodel comes in form of package hierarchies. The outermost level contains two packages: the *UML4SPM_Foundation* package and the *UML4SPM_Extensions* package (see figure 5.1). The *UML4SPM_Foundation* package contains UML2.0 Superstructure packages and concepts reused as a basis for defining mechanisms for activities and actions sequencing (e.g., control flows, object flows, etc), events, exception handling, constraint expressions, etc [OMG 07b]. The *UML4SPM_Extensions* package holds concepts in terms of MOF2.0-Compliant metaclasses with a proper semantics required for software process modelling. These concepts extend UML2.0 Superstructure metaclasses defined in the *UML4SPM_ Foundation*. The *UML4SPM_Extensions* package holds the *ProcessStructure* package and may contain any other packages that tend to extend UML4SPM for a specific purpose. As introduced earlier, since we aimed to participate at the OMG's SPEM revision, we have intentionally tried to keep the same naming conventions of the SPEM1.1's metamodel packages and metaclasses [OMG 05a].

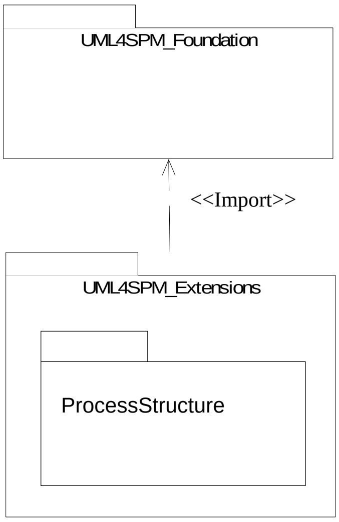


Figure 5.1. UML4SPM Metamodel Package Hierarchies

In the following we start the description of the UML4SPM metamodel by the *ProcessStructure* package, which is the building block of UML4SPM.

3.1. Process Structure Package

The *Process Structure* package is the core of UML4SPM. It introduces main concepts proper to software process modeling such as *Software Activity, Role*, *WorkProduct*, *Guidance*, etc. These concepts are presented in Figure 5.2, which represents the MOF-Compliant metamodel. *Process Structure* metaclasses will then extend UML2.0 Superstructure concepts we identified as basis of UML4SPM (c.f. Section 3.2.).

In the following, we give a detailed description of each of the metamodel's metaclasses. For each metaclass, we give its description, attributes, associations, and its direct generalizations and constraints if any. Metaclasses are introduced according to their importance and not in an alphabetical order.

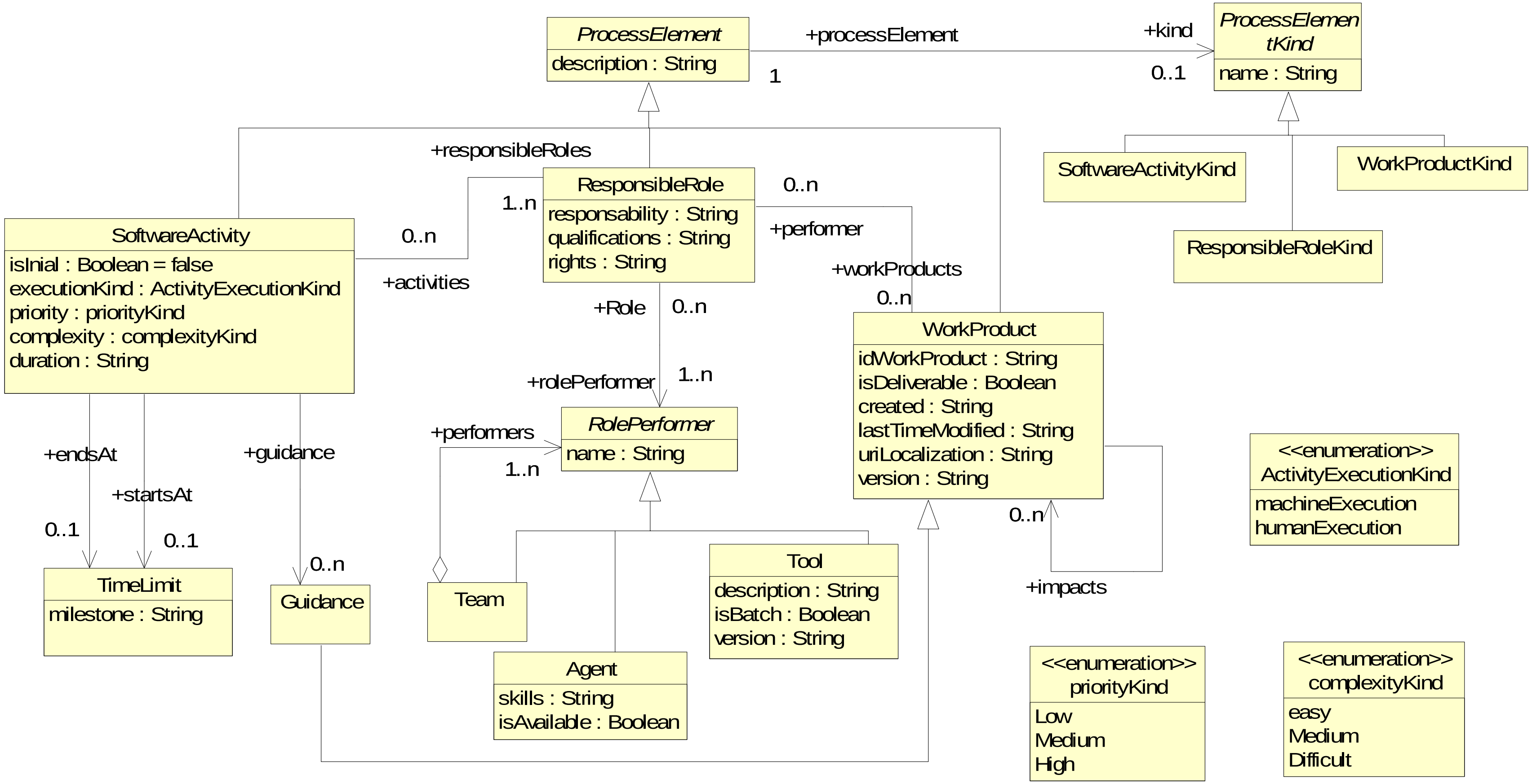


Figure 5.2. UML4SPM Process Structure Package

Process Element

Description

The abstract *Process Element* metaclass represents an abstraction of main process constituents i.e., *Software Activities*, *Responsible Roles* and *WorkProducts*. The introduction of this metaclass together with the *Process Element Kind* metaclass aims at providing an extension mechanism to the language in order to be adapted for a specific process or methodology domain. This mechanism is explained in more detail along this section and is illustrated in the next chapter.

Generalization

UML Superstructure::Kernel::Classifier.

Process Element extends the *Classifier* abstract metaclass. A *Classifier* is a namespace (i.e., can contain a set of named elements) and may have features which can be either structural (i.e., properties) or behavioural (i.e., operations). Thus, making a *Process Element* extending the *Classifier* metaclass allows process elements (i.e., *Software Activity*, *Responsible Role* and *WorkProduct*) to be enriched with new properties or operations additionally to the properties we already defined. This can be very helpful for adapting a process model to specific domain requirements.

Attributes

description: String It gives a description about the process element. For instance, in case of *Software Activity* this attribute will present main lines of the activity, its priority in the process, etc.

name: String (inherited from *Classifier*) Name of the process element

Other attributes inherited from UML Superstructure::Kernel::Classifier are given in more detail in [OMG 07b].

Associations

kind: Process Element Kind [0..1]

It is possible to define different kinds of process element instances. One process element may have zero or one kind. As an example, we can for instance define different kinds of *WorkProducts* such as Document, Code, Check List, Model, etc. When instantiating a *WorkProduct* within a process model, we can specify its kind (e.g., Model) among the set of *WorkProduct kinds* defined by the process modeller and which often can be process or methodology-specific. This association is redefined when used between *Process Element* subclasses and *Process Element Kind* subclasses.

Associations inherited from UML Superstructure::Kernel::Classifier are given in more detail in [OMG 07b].

Constraints

A *Process Element Kind* (i.e., *SoftwareActivityKind*, *WorkProductKind* and *ResponsibleRoleKind*) cannot be reused for different subclasses of the *Process Element* metaclass (i.e., *Software Activity, WorkProduct* and *Responsible Role*). It is applicable to only one subclass of *Process Element* or to its subclasses. A kind defined for a specific *WorkProduct* cannot be reused as a kind of a *Responsible Role* or a *Software Activity*.

To explicitly apply this constraint, we decided to redefine the "*kind*" association defined between *Process Element* and *Process Element Kind* metaclasses at the subclasses level of both metaclasses i.e., between *Software Activity*, *WorkProduct* and *Responsible Role* and respectively *SoftwareActivityKind*, *WorkProductKind* and *ResponsibleRoleKind* (see figure 5.3).

Other solutions are imaginable such as the definition of a property within the *ProcessElementKind* metaclass, which would be an enumeration of possible process element kinds (i.e., *SoftwareActivityKind*, *WorkProductKind* and *ResponsibleRoleKind*). However, this would lead to the writing of some OCL rules. One also can type the *ProcessElementKind* as a MOF *Class*. (i.e., create an association between the two metaclasses) At instantiation time, one applicable class among *Process Element* subclasses has to be selected. Here also, OCL rules are needed to make sure that the *Process Element Kind* is not applied to different kinds of *Process Elements* subclasses.

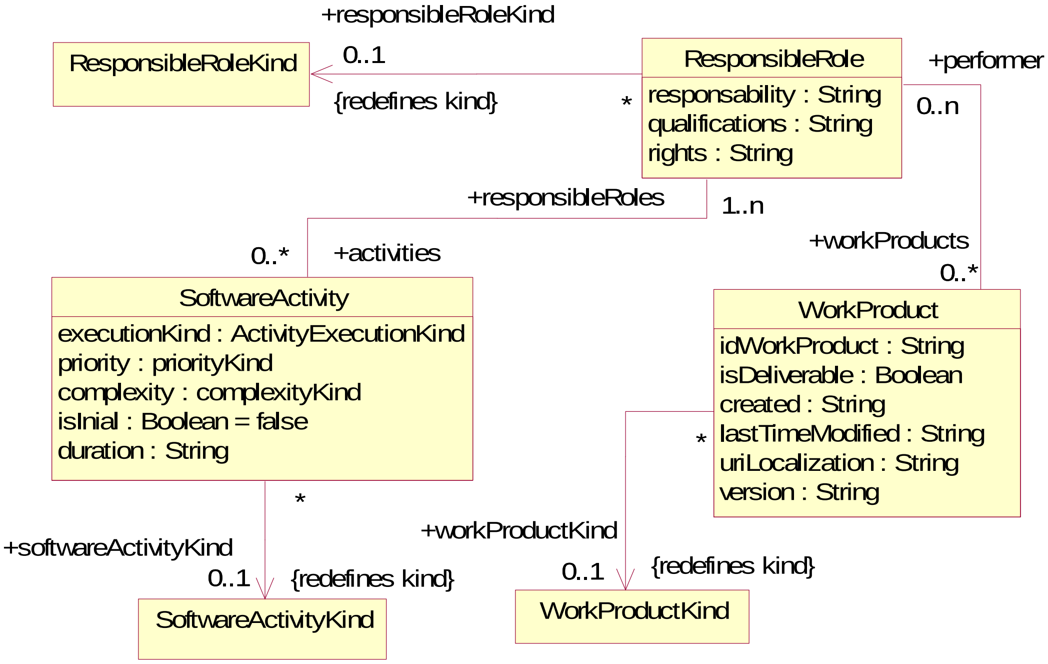


Figure 5.3. Redefinition of the "*Kind*" Association

Process Element Kind

Description

The abstract *Process Element Kind* metaclass is used to define process-specific or user-defined kinds of process elements (i.e., specific kinds of *WorkProducts*, specific kinds of *Responsible Roles*, etc.)

Generalization

None.

Attributes

name: String	It defines the name of the *Process Element* kind. Example of Kind's name: a "Model" in case of a *WorkProduct* kind, a "Phase" in case of *Software Activity*, etc.

Associations

None.

Constraints

None.

Software Activity Kind

Description

Defines user-defined or methodology and process-specific kinds of Software Activities.

Generalization

Process Element Kind

Attributes

name: String (from Process Element Kind)	It defines the name of the *Software Activity Kind*. Examples: "Phase", "Sprint", "Process", "Discipline", "Iteration", "Activity", etc.

Associations

None.

Constraints

None.

WorkProduct Kind

Description

Defines user-defined or methodology and process-specific kinds of WorkProducts.

Generalization

Process Element Kind

Attributes

name: String (from Process Element Kind)	It defines the name of the *WorkProduct Kind*. Examples: "Document", "Model", "Code", etc.

Associations

None.

Constraints

None.

Responsible Role Kind

Description

Defines user-defined or methodology and process-specific kinds of Responsible Roles.

Generalization

Process Element Kind

Attributes

name: String (from Process Element Kind)	It defines the name of the *Responsible Role Kind*. Examples: "Analyst", "Project Manager", "Designer", etc.

Associations

None.

Constraints

None.

Software Activity

Description

The building block of any UML4SPM process model is the *Software Activity* element. It describes any effort or piece of work to be performed during the software development process. It has a *name* and a *description* property (inherited from *Process Element*) that briefly outlines what has to be done by *Responsible Roles* of the activity, a *priority* ranging from "low" to "high" to highlight its importance within the process and a *complexity* property to show its degree of difficulty (i.e., easy, medium and difficult). For instance, a *Software Activity* with *priority* set at "medium" and *complexity* set at "high" would imply people that are more skilled and fewer rigors regarding the schedule, tests and resource allocations.

A *Software Activity* may have a *Kind*, which can be user-defined or process specific. Examples of *Software Activity Kinds* could be a "Process", an "Activity", a "Phase", a "Discipline" (in RUP for instance), a "Sprint" (in the Scrum agile process), "Iteration", etc.

The *isInitial* property is to tell whether the activity is the initial one within the process or not. This is very crucial in the sense that, in runtime, the initial activity is treated differently than the other activities. A special behavior is assigned to it and it is considered as the current context of the process (i.e., the process containing all other sub-processes or *Software Activities*). Thus, any UML4SPM process model should have an outermost *Software Activity* with its *isInitial* property set at "true" and, which encapsulates or invokes all subsequent activities. The idea is to have one metaclass to represent a process, a sub-process, a phase, a *Software Activity*, etc. This aims at facilitating the reusability of *Software Activitie*s for building processes that are more complicated.

A *Software Activity* can be totally executed by a machine. Then, the *ActivityExecutionkind* property is set to "machine execution". Otherwise, it is fixed at "human execution" if a human expertise is required. The possibility to express *Software Activity* milestones is given thanks to the *TimeLimit* metaclass. This helps in defining the starting time and ending time of an activity, a very useful option for process monitoring. Otherwise, it is possible to simply specify the *Software Activity's* duration through the *duration* property. Finally, a *Guidance* may be needed to realise the activity.

Software Activity Lifecycle

A Software Activity being indirectly a *Classifier* (cf. Section 3.2.) and to formally determine the *Software Activity* lifecycle, we decided to explicitly provide it with a predefined set of states (i.e., *Initialized, Assigned, Running, Suspended, Terminated, and Aborted*). These states can be reached thanks to built-in methods we defined for each *Software Activity* as it is shown in figure 5.4. This state diagram can be used in defining an execution engine for UML4SPM and is inspired from works done in [Di Nitto 02] and [Dami 98]. More details of *Software Activity* built-in methods are introduced in Chapter 8.

First, when an instance of the *Software Activity* is created its state is fixed at *"Initialized"* and waits for an agent's agreement to perform the task. *Role Performers* suitable for handling the activity are selected according to the *Responsible Role* qualifications in charge of the activity. A matching is performed between qualifications required to undertake the role and agent skills. When the *Software Activity* instance is assigned to agents susceptible to take in charge the activity, the *"Assigned"* state is reached. If an agent accepts the responsibility of performing the work, the *Software Activity* enters the *"Running"* state. While running, the *Software Activity* can be *"Suspended"* for some reasons (e.g., schedule, agent availability, project manager order, etc.), or *"Aborted"* if any problems or by order of the project manager. Finally, when the *Software Activity* is completed with success, the *"Terminated"* state is reached. The *"Aborted"* state can be reached from any state.

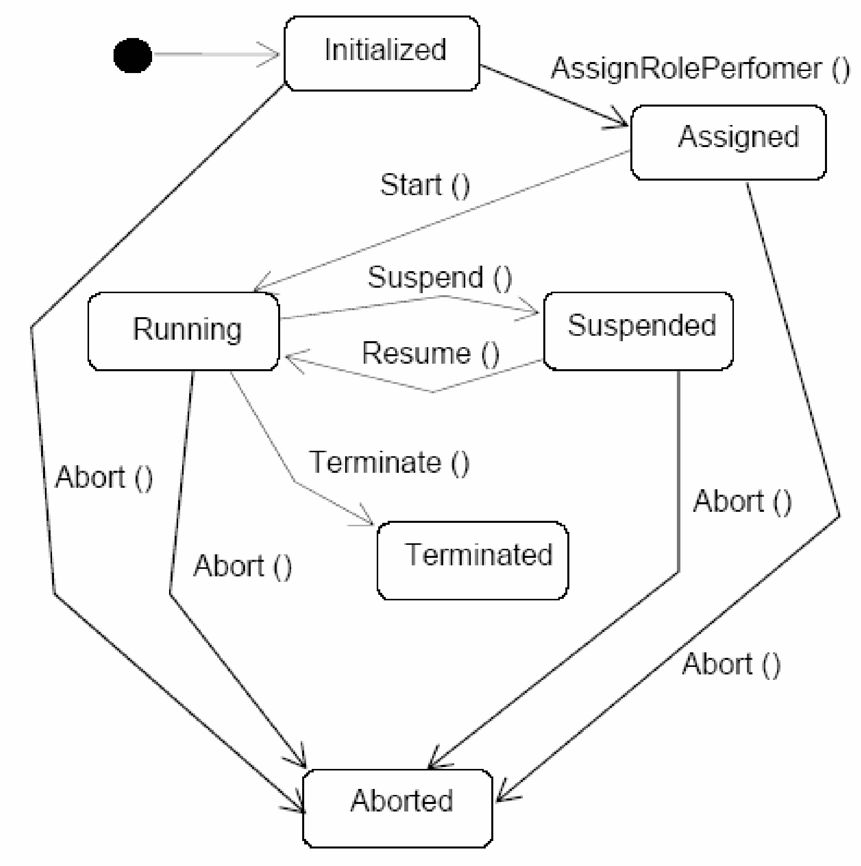


Figure 5.4. UML4SPM Software Activity Lifecycle

Generalization

Process Element

UML Superstructure::Activities::Activity

A *Software Activity* being a *Process Element*, process modellers can define their process or methodology-specific kinds of *Software Activity*. A *Software Activity* extends the UML2.0 Superstructure *Activity* (cf. Section 3.2.1 bellow).

Attributes

name: String (from *Classifier*)	Name of the *Software Activity* (or of the Process, Phase, etc depending on its *Kind*)
description: String (from *Process Element*)	Gives a brief description of the *Software Activity*
executionKind: ActivityExecutionKind	Defines whether the *Software Activity* execution is fully automated or has to be executed by a human
priority: priorityKind	Specifies the activity's priority within the process (i.e., low, medium or high)
complexity: complexityKind	This information helps in assigned the appropriate skills and resources to the activity depending on its degree of complexity
isInitial: Boolean	Specifies whether the activity is the initial one within the process or not. If *isInitial* is set to "true", the software activity is considered as the container and context of all other sub-activities.
duration: String	Duration of the activity in terms of days

Associations

responsibleRoles: ResponsibleRole[1..n]	A *Software Activity* may have one or more *Responsible Roles* in charge of performing the activity
softwareActivityKind: SoftwareActivityKind [0..1]	Defines the kind of the *Software Activity*. This association redefines the *ProcessElement::kind* association
requires: Guidance [0..n]	For performing a *Software Activity* zero or more Guidance may be required such as guidelines, tool tutorials, check lists, etc.
startsAt: TimeLimit [0..1]	A process modeller can affect a start time and an end time in order to control the process schedule and activities monitoring
endsAt: TimeLimit [0..1]	Specifies the end time of a *Software Activity*

Constraints

In a UML4SPM process model, it is required that one and only one *Software Activity* has its *isInitial* property set to "true". That latter is considered as the process context. The OCL rule for this constraint is as follow:

```
      context Model inv:
self.allOwnedElement()->select(oclIsKindOf(SoftwareActivity))->select(sa | sa.isInitial)->size()=1
```

Activity Execution Kind

Description

Activity Execution Kind is an enumeration defining the possible kinds of executing a *Software Activity*. The literal values of this enumeration are:

- machineExecution: for a *Software Activity* which is completely executable by a tool, a software or a service.
- humanExecution: for any *Software Activity* which requires agents involvement.

Generalization

None.

Attributes

None.

Associations

None.

Constraints

None.

Priority Kind

Description

The *Priority Kind* enumeration defines the *Software Activity's* priority within a process. There are some activities within a process, which are strategic and may require more attention at performing time. The literal values of this enumeration are:

- Low
- Medium
- High

Generalization

None.

Attributes

None.

Associations

None.

Constraints

None.

Complexity Kind

Description

The *Complexity Kind* enumeration defines the *Software Activity's* priority within a process. Activities with complexity set to "Difficult" would require people that are more skilled or may need more time than those set to "Easy". The literal values of this enumeration are:

- Easy
- Medium
- Difficult

Generalization

None.

Attributes

None.

Associations

None.

Constraints

None.

Responsible Role

Description

Responsible Roles are also important constituents of UML4SPM process models. The *Responsible Role* metaclass defines *responsibilities* and *qualifications/skills* required for performing a *Software Activity*. At process execution time, *Responsible Roles* are undertaken by *Role Performers*, which can be an *Agent*, a *Team* of a *Tool*. A *Responsible Role* is also responsible of the *WorkProducts* realised within the process. It is possible to define *Kinds* of *Responsible Roles* s. Examples of such role kinds would be "Analyst", "UML Modeller", "Java Tester", "Designer", "Project Manager", etc. At instantiation time, the *Responsible Role* instance can be assigned with one of these role *Kinds*.

Generalization

Process Element

A *Responsible Role* being a *Process Element*, process modellers can define their process or methodology-specific kinds of *Responsible Roles*.

Attributes

name: String (from *Classifier*)	Name of the *Responsible Role*
description: String (from *Process Element*)	A description about what it is expected from the *Responsible Role*, some guidelines, important characteristics, etc.
responsibility: String	A *Responsible Role* has some responsibilities regarding the performing of a *Software Activity* or a *WorkProduct*. They are given here
qualification: String	Defines the qualifications required from a *Responsible Role*.

	Examples of such qualifications would be: UML Design, Java programming and testing, SQL, QVT, etc.
rights: String	In some cases or in some information systems, roles may have a limited access to process's activities or a restricted manipulation of process's workproducts (i.e., only in reading, not in modification, etc.)

Associations

activities: SoftwareActivity [0..n]	The *Software Activities* the *Responsible Role* is in charge. *A Responsible Role* may be assigned to zero or more *Software Activities*
workproducts: WorkProducts [0..n]	A *Responsible Role* may have the responsibility of zero or more *WorkProducts*. It is in charge of delivering them on time and in respect with the project or application requirements
responsibleRoleKind: ResponsibleRoleKind [0..1]	Defines the kind of the *Responsible Role*. This association redefines the *ProcessElement::kind* association
rolePerformer: RolePerformer [1..n]	A *Responsible Role* may be undertaken by one or more *RolePerformers*. At execution time, a matching is done between qualifications required for a *Responsible Role* and *Role Performer's skills* and knowledge. *Role Performers* that satisfy those qualifications are then proposed to the project manager for selection and assignment

Constraints

None.

WorkProduct

Description

WorkProduct represents any physical piece of information consumed, produced or modified during the software development process. It has a *name*, a *description* and may be under the responsibility of zero or more *Responsible Roles*. A *WorkProduct* may be composed of other *WorkProducts* and the modification of one *WorkProduct* may affect other *WorkProducts*. This is indicated thanks to the *impacts* reflexive association. A *WorkProduct* has a unique identifier specified by the *idWorkProduct* property. A *WorkProduct* can be a process deliverable or not, this is indicated thanks to the *isDeliverable* property. The *uriLocalization* property serves at determining the *WorkProduct* location during process execution. The *created*, *version* and *lastTimeModified* properties were introduced in order to help developers in avoiding confusion while manipulating different versions of the same *WorkProduct* during development activities

A *WorkProduct* may have a *Kind*, which can be for instance, "Code", "Document", "Model", etc. these *Kinds* are to be defined by process modellers according to a process or methodology domains.

Finally, *WorkProducts* are used as inputs / outputs of *Software Activities* and of atomic actions within *Software Activities*. However, this facility is not directly supported by the MOF-compliant metamodel we propose, but will be introduced by extending UML2.0 Superstructure *Activity* concepts. This facility is addressed in more detail in Section 3.2.

Generalization

Process Element

UML Superstructure::Deployments::Artifacts::Artifact

A *WorkProduct* being a *Process Element*, process modellers can define their process or methodology-specific kinds of *WorkProducts*. The *WorkProduct* metaclass extends the UML2.0 Superstructure *Artifact*. This is addressed in more detail in Section 3.2.2 hereunder.

Attributes

name: String (from *Classifier*)	*WorkProduct* name
description: String (from *Process Element*)	Gives a description about the *WorkProduct* to be used/produced by the *Software Activity* or by one of its atomic actions (steps)
idWorkProduct: String	A unique identifier of the *WorkProduct* within the software development process
isDeliverable: Boolean	Not all *WorkProducts* used within the process are deliverables. It is important to distinguish between transient artifacts and deliverables, which represent the result of the development process. A deliverable requires more attention from process participants than simple artifacts. A *WorkProduct* can start the process as no deliverable and ends as a process deliverable
created: String	*WorkProduct* creation time
lastTimeModified: String	Specifies the last time the *WorkProduct* was modified.
version:String	Gives the version of the *WorkProduct*. In case of tool documentations for instance or a language guidelines, checklists, Code classes, libraries, etc. it is important to make sure that all process participants are using the right version of the *WorkProduct*.
uriLocalisation: String	Specifies *WorkProduct* localization at run time

Associations

performer: ResponsibleRole [0..n]	Refers to *Responsible Roles* in charge of the

	WorkProduct
workProductKind: WorkProductKind [0..1]	Defines the kind of the *WorkProduct*. This association redefines the *ProcessElement::kind* association
impacts: WorkProduct [0..n]	A modification of one *WorkProduct* may affect other *WorkProducts*.

Constraints

None.

Role Performer

Description

A *Role Performer* is an abstract metaclass and represents the entity that may undertake a *Responsible Role* in order to perform a *Software Activity*. Subclasses of *Role Performer* are *Team*, *Agent* and *Tool*

Generalization

None.

Attributes

name:String	Name of the *Role Performer*

Associations

None.

Constraints

None.

Tool

Description

A *Tool* is defined by its *name*, a *description* (e.g., a link to online tutorials), an *isBatch* property if the tool is to be used in batch or in user-interface mode and a *version* number. That latter can turn out to be very helpful especially during the design and implementation phases of the software development process. Indeed, source codes may be handled under different tool versions (e.g., Compiler version 1.4), which may be confusing for developers. The *version* property should avoid these conflicting situations.

Generalization

Role Performer.

Attributes

name (from Role Performer): String	Name of the *Tool*
description: String	A description on how to use the tool, the facilities it provides, some guidelines or a web link to online tutorials, etc.
isBatch: Boolean	Indicates if the *Tool* is to be used in batch or through a GUI

version: String	Indicates the version of the *Tool* used in performing the *Software Activity* or in handling the *WorkProducts*.

Associations

None.

Constraints

None.

Team

Description

A *Team* is *Role Performer* in charge of undertaken a *Responsible Role* in order to perform one or more process's activities. A *Team* can be composed by *Agents* or by other *Teams*.

Generalization

Role Performer.

Attributes

name: String (From *RolePerformer*)	Team's name

Associations

performers: RolePerformer [1..n]	A *Team* can be composed of one or more *Agents* or *Teams*.

Constraints

A *Team* can only be composed by *Agents* or by other *Teams*. A *Team* cannot be composed of *Tools*. Hereunder the OCL rule corresponding to this constraint.

```
context Team inv:
self.performers->forAll (roleperformer |roleperformer.isKindOf (Team) or roleperformer.isKindOf(Agent)
```

Agent

Description

Agent is the human that may undertake a *Responsible Role* in order to realize a *Software Activity*. It may also have the responsibility of some *WorkProducts*. An *Agent* can be part of a zero or more *Teams*.

Generalization

Role Performer.

Attributes

name: String (From *RolePerformer*)	*Agent*'s name
skills: String	Represents the *Agent's* skills. These skills have to be compared with qualifications required from a *Responsible Role*. If they match, the *Agent* will be proposed as a potential *Role Performer*
isAvailable: Boolean	Indicates the availability of the Agent.

	Required when assigning *Agents* to *Responsible Roles*.

Associations

None.

Constraints

None.

Guidance

Description

In order to perform process activities, some guidelines are required. They help in understanding the work to be done, give some hints and tips for a better comprehension of the *Responsible Role* in charge of realizing the activity. In the software process development discipline, this is called *Guidance*. Many *Kinds* of *Guidance* can be defined depending on the methodology or process followed for building software.

Generalization

WorkProduct.

Attributes

See the *WorkProduct* metaclass for attributes.

Associations

None.

Constraints

None.

Time Limit

Description

Represents the time at which a *Software Activity* may start or has to finish.

Generalization

None.

Attributes

milestone: String	Indicates the *Start* or *End* time of a *Software Activity*

Associations

None.

Constraints

None.

The UML4SPM metaclasses we introduced in the *Process Structure* package represent the set of constructs and semantics required for modeling primary elements of software process models. However, this set is incomplete. Coordination of *Software Activities* (i.e., control and data flows), the ability to express events, decisions, constraints, iterations, exceptions, and actions with operational semantics is still lacking. This is where the *UML4SPM Foundation* package comes into action.

3.2. UML4SPM Foundation Package

In this package, we regroup the set of UML2.0 Superstructure concepts we identified as a basis of UML4SPM. As we stated earlier, the *Process Structure* package introduces only the set of concepts proper to software process modeling. Concepts proper to the sequencing of activities, synchronization, event and exception handling and more important, the possibility to define actions with an executable semantics are provided by the *UML4SPM Foundation* package. Indeed, instead of reinventing the wheel, we privileged reusing the expressiveness of UML2.0 *Activity* and *Action* packages, which show a high potential for modeling processes [Störrle 04] [Vitolins 05] [OMG 07b].

In addition, as we highlighted it in Section 2 (cf. Design Goals), since we aimed to participate in the OMG's SPEM standard revision, we had the constraint to respect RFP's mandatory requirements, which advocated reusing UML2.0 *Activity* diagrams as a basis [OMG 04].

The UML4SPM Foundation package contains not only metaclasses and packages required for defining *Activity* diagrams (represented as shaded boxes in figure 5.5), but also their direct and indirect super metaclasses. Figure 5.5 gives an overview of the UML2.0 Superstructure packages needed to define *Activity* diagrams. However, in the following we will introduce only UML2.0 Superstructure metaclasses we extended or we reused within UML4SPM. More details on metaclasses not addressed here as well as their semantics are given in [OMG 07b].

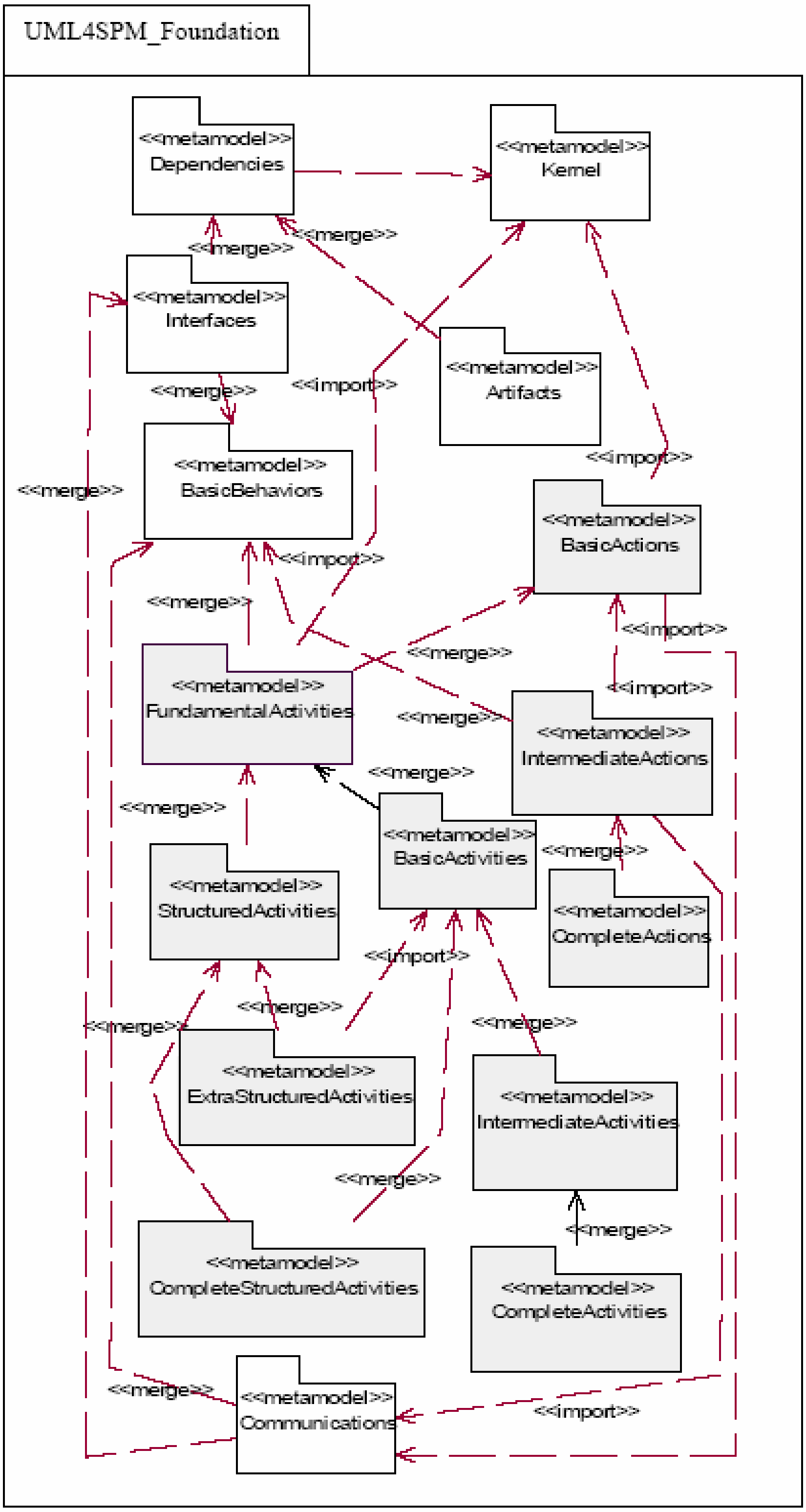

UML4SPM_Foundation
<<metamodel>> Dependencies
<<metamodel>> Kernel
<<merge>>
<<merge>>
<<metamodel>> Interfaces
<<metamodel>> Artifacts
<<merge>>
<<import>>
<<metamodel>> BasicBehaviors
<<merge>>
<<import>>
<<metamodel>> BasicActions
<<import>>
<<merge>>
<<import>>
<<merge>>
<<metamodel>> FundamentalActivities
<<import>>
<<merge>>
<<metamodel>> IntermediateActions
<<merge>>
<<merge>>
<<metamodel>> BasicActivities
<<merge>>
<<metamodel>> StructuredActivities
<<metamodel>> CompleteActions
<<import>>
<<merge>>
<<merge>>
<<merge>>
<<metamodel>> ExtraStructuredActivities
<<metamodel>> IntermediateActivities
<<merge>>
<<merge>>
<<metamodel>> CompleteStructuredActivities
<<metamodel>> CompleteActivities
<<merge>>
<<metamodel>> Communications
<<import>>

Figure 5.5. UML4SPM Foundation Package

3.2.1. Activity

In UML2.0, an *Activity* is the specification of a parameterized behavior defined in terms of a coordinated sequencing of *Actions* [OMG 07b]. The sequencing of these actions is ensured using an *Object* and *Control* flow model. The former is used to sequence data produced by one action that are used by other actions (e.g., data outputs of action A are to be used as B's inputs). The latter is used to explicitly sequence the execution of actions (e.g., action B starts when A finishes). *Actions* within activities may be initiated because other behaviors in the model terminate, because objects and data become available, or because events occur external to the flow. UML2.0 defines various kinds of actions, which vary as follows:

- Occurrences of primitive functions;
- Invocations of behavior (e.g., *CallBehaviorAction, CallOperationAction*);
- Communication actions, such as the sending of *Signals*;
- Handling of objects, such as the reading or writing of attributes or associations.

Activities also include *Control Nodes*, which structure control and object flow between actions. In addition to *Initial* and *Final* Nodes, these include *Decision Node* to express choices, *Fork Node* for expressing concurrency (parallelism), *Join Node* for synchronization and *Merge Node* to accept one among several alternate flows.

Object Nodes in activities are to represent objects and data as they flow in and out of invoked behaviors.

As we made the UML4SPM *Software Activity* element extending the UML2.0 *Activity*, we take advantage of all its properties, associations and capabilities (see figure 5.6). Thus, a *Software Activity* can be composed by other *Software Activities* (capability inherited -indirectly- from *Classifier*) and may contain *Actions* (cf. Section 3.2.3), *Object Nodes (i.e., Pins, Activity Parameter Nodes, etc.) and Control Nodes (i.e., Fork, Join, Merge, Decision nodes, etc.). Actions* will be then considered as the atomic components (steps) of a *Software Activity*.

A UML2.0 *Activity* being indirectly a *Classifier*, the possibility to specify new properties and new operations is then offered to *Software Activities*. Thus, the process modeller can customize the definition of process *Software Activities* by adding new properties depending on process domains (e.g. to add the *weight* property to express that a *Software Activity* represents 10%, 20% or more of the entire process), define new operations and helpers or specify composite activities. A *Software Activity* being now a specialization of UML2.0 *Activity* metaclass, the specification of pre and post conditions on the execution of a *Software Activity* is also rendered possible. (e.g., Post-Condition: the activity's WorkProduct output state ="Validated").

Semantics of the UML2.0 *Activity* is given in more details in [OMG 07b] and will be addressed further in this document for the purpose of UML4SPM process model executions (cf. Chapter 8). In the following, we give generalisations, attributes, associations, and constraints of the UML2.0 *Activity* metaclass, which are inherited by the UML4SPM *Software Activity* metaclass. The subset of *Activity* diagram elements (i.e., *Control Nodes, Object Nodes, Flows*, etc.) we reused in UML4SPM are depicted at the end of this chapter in figure 5.9.

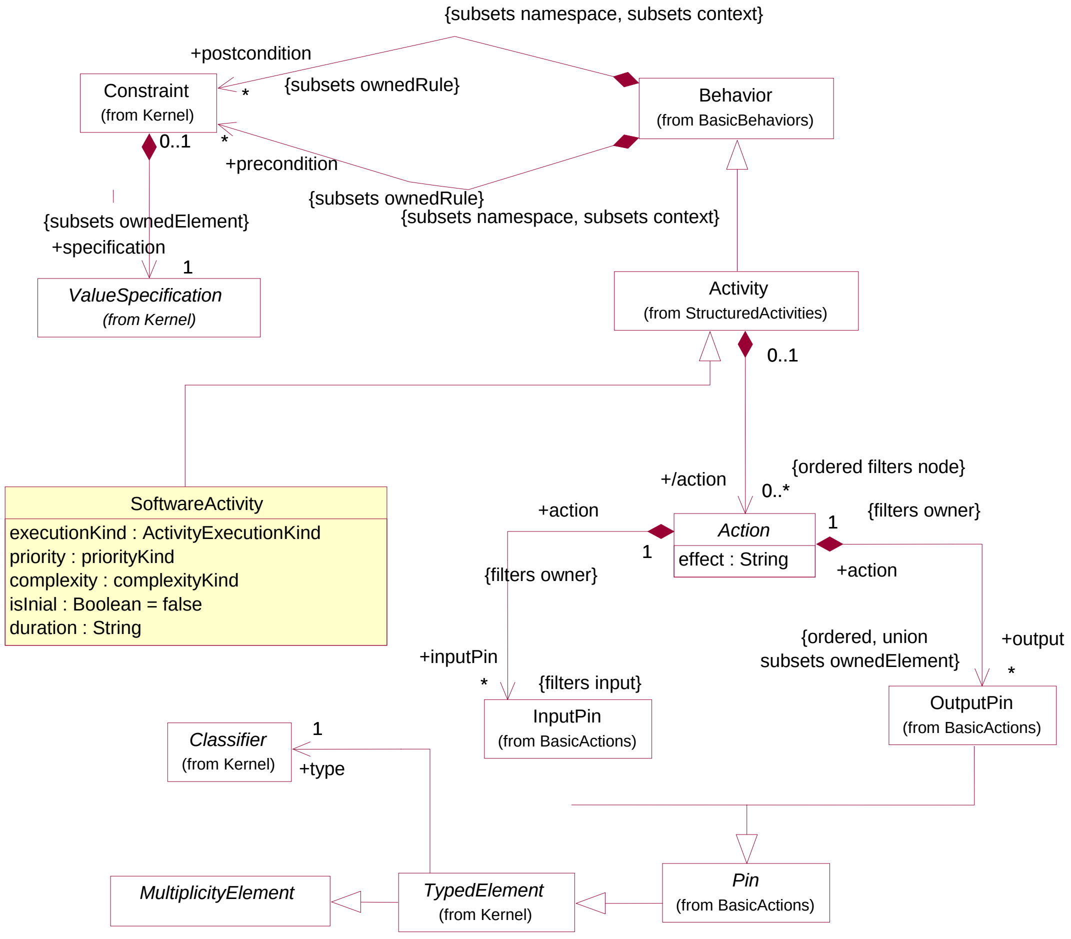


Figure 5.6. UML4SPM Software Activity extending UML2.0 Activity

Generalization

Behavior (from BasicBehaviors)

More details on the Behavior metaclass can be found in [OMG 07b]

Attributes

isReadOnly : Boolean = false (From Basic Activities)	If *true*, this activity must not make any changes to variables outside the activity or to objects. (This is an assertion, not an executable property. It may be used by an execution engine to optimize model execution. If the assertion is violated by the action, then the model is ill formed.) The default is false (an activity may make non-local changes).
isSingleExecution : Boolean = false (from Complete Activities)	If *true*, all invocations of the activity are handled by the same execution.

Associations

group: ActivityGroup [0..*] (from Fundamental Activities)	Top-level groups in the *Activity*.. *Activity Groups* are a generic grouping construct for *Activity* nodes and edges.
node: ActivityNode [0..*] (from Fundamental Activities)	Nodes coordinated by the activity. {Subsets *Namespace::ownedElement*}. Subclasses of *Activity Node* are *Actions*, *Object Nodes*(e.g., Pins, Activity Parameter Nodes, etc.) and *Control Nodes* (e.g., Fork, Merge, Join, etc)
edge: ActivityEdge [0..*] (from Basic Activities)	Edges expressing flow between nodes of the activity. {Subsets Namespace::ownedElement}. The expression of Guards on Edges is possible. Guards are evaluated at runtime to determine if the edge can be traversed.
partition: ActivityPartition [0..*] (from Intermediate Activities)	Top-level partitions in the activity
structuredNode: StructuredActivityNode [0..*] (from Structured Activities)	Top-level structured nodes in the activity. A structured activity node represents a structured portion of the activity that is not shared with any other structured node, except for nesting
variable: Variable [0..*] (from Structured Activities)	Top-level variables in the activity

Constraints

[1] The nodes of the activity must include one *ActivityParameterNode* for each parameter.

[2] An activity cannot be autonomous and have a classifier or behavioural feature context at the same time.

[3] The groups of an activity have no super groups.

3.2.2. Artifact

The UML2.0 standard defines an *Artifact* as a *Classifier* that represents a physical entity. It may have properties that represent its features, and operations that can be performed on its instances. It can be involved in associations to other *Artifacts* (e.g., composition associations). Examples of *Artifacts* include model files, source files, scripts, and binary executable files, a development deliverable, etc.

The UML4SPM *WorkProduct* element extends UML2.0 *Artifact*. An *Artifact* being a *Classifier*, *WorkProducts* can be defined as parameters of *Software Activities* in terms of *Activity Parameter Nodes* and as inputs/outputs of *Actions* in terms of *InputPins* and *OutputPins*. They can also have additional properties and operations than those we explicitly defined. It is possible to specify composite *WorkProducts* thanks to the "nested artifact" association, the set of model elements that are utilized in the construction of the *WorkProduct* through the "manifestation" association (see figure 5.7). Finally, a *WorkProduct* may be associated with a state machine that defines its allowable states and operations to switch between these states. Contrarily to a *Software*

Activity, *WorkProducts* do not have predefined states and built-in methods. It is up to the process modeller, depending on the process domain, to define them at design time if needed.

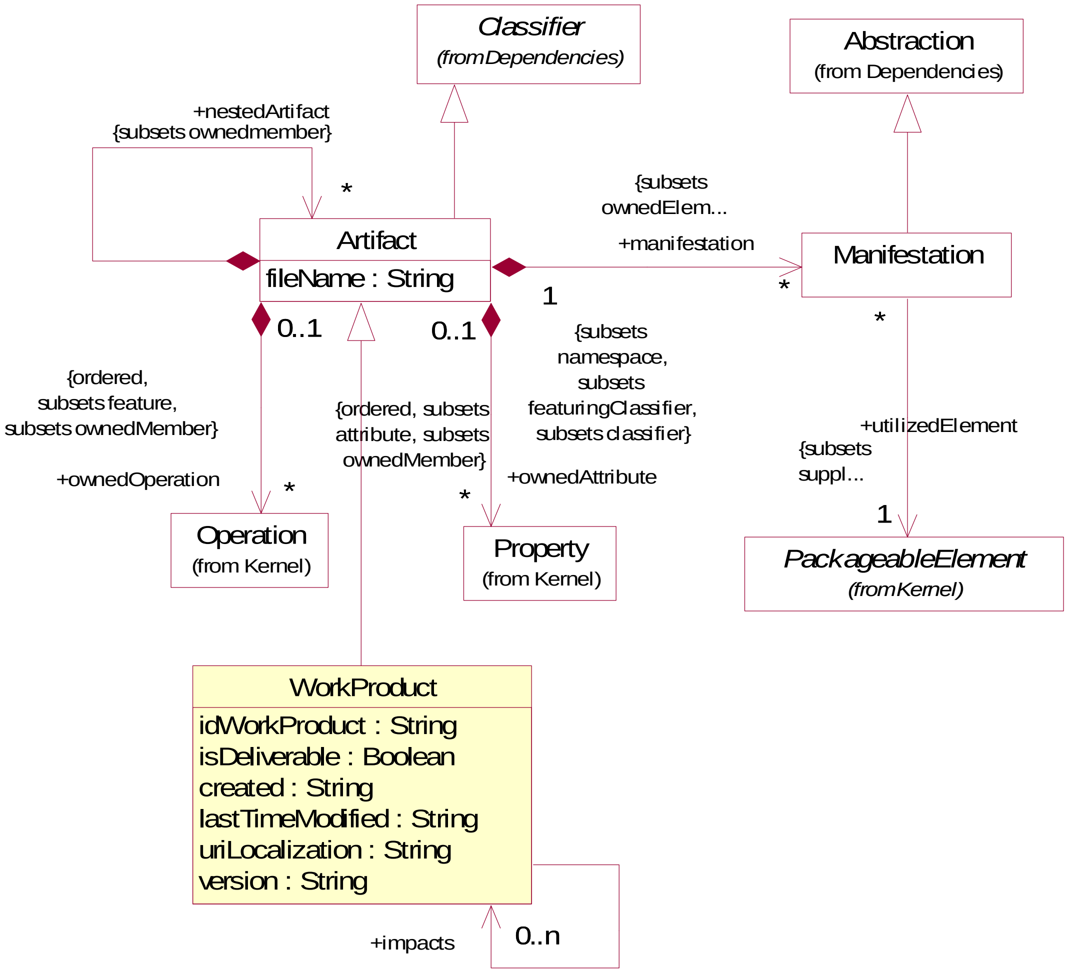


Figure 5.7. UML4SPM WorkProduct extending UML2.0 Artifact

Generalization

- *Classifier* (from Kernel, Dependencies, PowerTypes)
- *DeployedArtifact* (from Nodes)
- *NamedElement* (from Kernel, Dependencies)

More details on these metaclasses can be found in [OMG 07b]

Attributes

fileName: String — A concrete name that is used to refer to the *Artifact* in a physical context. Example: file system name, universal resource locator, etc.

Associations

nestedArtifact: Artifact [*] — The *Artifacts* that are defined (nested) within the *Artifact*. The association is a specialization of the

	ownedMember association from *Namespace* to *NamedElement*.
ownedAttribute : Property [*]	The attributes or association ends defined for the *Artifact*. {Subsets *Namespace::ownedMember*}
ownedOperation : Operation [*]	The operations defined for the *Artifact*. {Subsets *Namespace::ownedMember*}
manifestation : Manifestation [*]	The set of model elements that are manifested in the *Artifact*. That is, these model elements are utilized in the construction (or generation) of the artifact. {Subsets *NamedElement::clientDependency*, Subsets *Element::ownedElement*}

Constraints

None.

3.2.3. Actions

The UML2.0 Superstructure standard defines an *Action* as the fundamental unit of behavior specification [OMG 07b]. An *Action* takes a set of inputs (called *Input Pins*) and converts them into a set of outputs (*Output Pins*), though either or both sets may be empty.

To express most semantics of executable actions that can be found in programming languages, UML2.0 offers four *Actions* packages (shaded boxes in figure 5.8.):

- ***Basic Actions*****:** This package includes what it is called *Invocation Actions*. This regroups actions that perform operation calls (i.e., *CallOperationAction*), signal sends (i.e., *SendSignalAction*), and direct behavior invocations (i.e., *CallBehaviorAction*). The *CallBehaviorAction* may be used within an *Activity* in order to call (synchronously/asynchronously) another *Activity*. The *Opaque Action* is also defined in this package, which represents an action with implementation-specific semantics.
- ***Intermediate Actions*****:** This package introduces various primitive actions. This includes actions for accessing object's structural features (i.e., *ReadStructuralFeatureAction*), for object and link creations/destructions (i.e., *CreateObjectAction*, *CreateLinkAction*, *DestroyObjectAction*, etc). Besides, more invocation actions are defined for broadcasting signals to the available "universe" and transmitting objects that are not signals (i.e., *BroadcastSignalAction*).
- ***Complete Actions*****:** Defines additional actions dealing with the relation between object and class and link objects. In addition, in this package, actions are defined for accepting events (i.e., *AcceptEventAction*), including operation calls (i.e., *AcceptCallAction*), and retrieving the property values of an object all at once, etc.
- ***Structured Actions*****:** These actions operate in the context of *Activities* and *Structured Nodes*. Variable actions support the reading and writing of variables. Variable actions can only access variables within the activity of which the action is a part. An action is defined for raising exceptions (i.e.,

RaiseExceptionAction) and a kind of input pin is defined for accepting the output of an action without using flows (i.e., *ActionInputPin*).

In the following we introduce the set of Actions we reuse in UML4SPM.

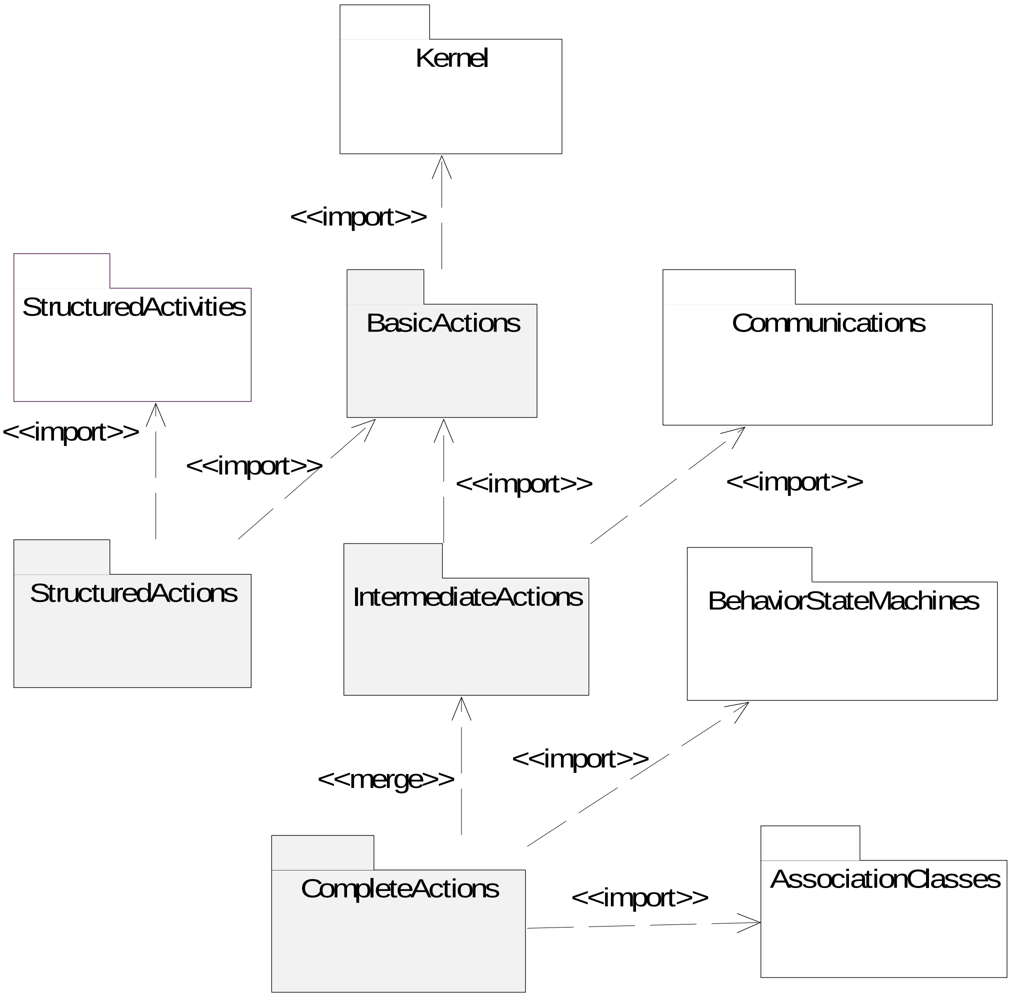


Figure 5.8. UML2.0 *Actions* Packages

UML2.0 Actions reused within UML4SPM

In the context of UML4SPM, we identified the set of UML2.0 *Actions* that can be reused for software process modeling. This set regroups actions that provide our language with powerful capabilities to express *Proactive* and *Reactive* controls. This encompasses actions for calling activities and operations, sending events, raising exceptions, etc. For more flexibility, we also opted for reusing the *Opaque Action* as a means to specify implementation-specific actions within process descriptions.

Besides, we believe it is not useful to support fine-grained actions provided by the standard. Examples of such actions include reading and writing of structural features, link creations, object destructions, and so on. These actions are too low-level and deals with object memory access, primitive functions, etc. [OMG 07b]. Using them within UML4SPM would make process models too complex and unreadable. In addition, a main software process characteristic is that they are too human-oriented. Thus, the only actions that can be automated are those needed to ensure process's activities coordination and sequencing, WorkProducts routing, roles affectations, etc (cf. Chapter 2, Section 4.1).

In the following, we briefly introduce the actions we reuse in UML4SPM. More details can be found in [OMG 07b]. Their notations are depicted in Appendix A of this document. Their execution semantics and implementations are given in Chapter 8.

CallBehaviorAction

Probably the action that we will use the most since it represents the mechanism by which an activity calls (initiates) another activity (in more general, a behavior). Calls may be synchronous or asynchronous. For synchronous calls the execution of the call behavior action waits until the execution of the invoked behavior completes and a result is returned on its output pin. The action completes immediately without a result, if the call is asynchronous. The call may have arguments which are specified in the action's input pins.

CallOperationAction

CallOperationAction is an action that transmits an operation call request to the target object. Since an activity is indirectly a *Classifier,* it can own *Operations*. These *Operations* can be invoked synchronously or asynchronously thanks to the *CallOperationAction*. This capability is very attractive since that a process modeller can define within activities, operations with computational instructions. These instructions can be defined using the UML behavior model or a specific implementation language such Java. The call may have arguments, which are typed.

SendSignalAction

It represents the way for defining reactive controls within UML4SPM. The *SendSignalAction* creates a *Signal* instance and sends it to a specific activity execution. This action can be used within process's activities as means to generate events that can be caught by other activities participating in the process. This action is to be used with the *AcceptEventAction* that represents the receptacle that will catch the generated signal. A variant of *SendSignalAction* is *SendObjectAction*, which transmits an object - whatever its type (and not only an instance of *Signal*) - to another object. The instance of the object is already created when the action is executed (has to be created in case of *SendSignalAction*).

AcceptEventAction

This action complements the *SendSignalAction*. It waits for the occurrence of an event meeting a specified condition in order to trigger. If the accept event action is executed and the object (in our case, an instance of *Activity*) detected an event occurrence matching one of the triggers on the action, then the *AcceptEventAction* completes and outputs a value describing the occurrence. UML2.0 defines three types of events. *Time Event, Change Event* and *Message Event*. The former specifies a point in time. At the specified time, the event occurs. A *Change Event* models a change in the system configuration that makes a condition true. Finally, a *Message Event* specifies the receipt by an object of either a call or a signal.

BroadcastSignalAction

This action is very useful in case of sending a signal to all activity instances (objects in general). However, the manner of identifying the set of objects that are broadcast targets is a semantic variation point and may be limited to some subset of all the objects that exist.

RaiseExceptionAction

In case of an unexpected situation during the activity execution, a *RaiseExceptionAction* occurs indicating the exception type. All flows within the activity are stopped and the appropriate handler is assigned.

OpaqueAction

OpaqueAction is introduced for implementation-specific actions or for use as a temporary placeholder before some other action is chosen. In the context of UML4SPM, we can imagine using this action for modeling manual actions or human interactions.

Figure 5.9 regroups the subset of most significant UML2.0 Superstructure *Activity* and *Action* concepts we reused in UML4SPM. Their semantics and their implementation are addressed in Chapter 8.

In the next section we present the UML4SPM notation.

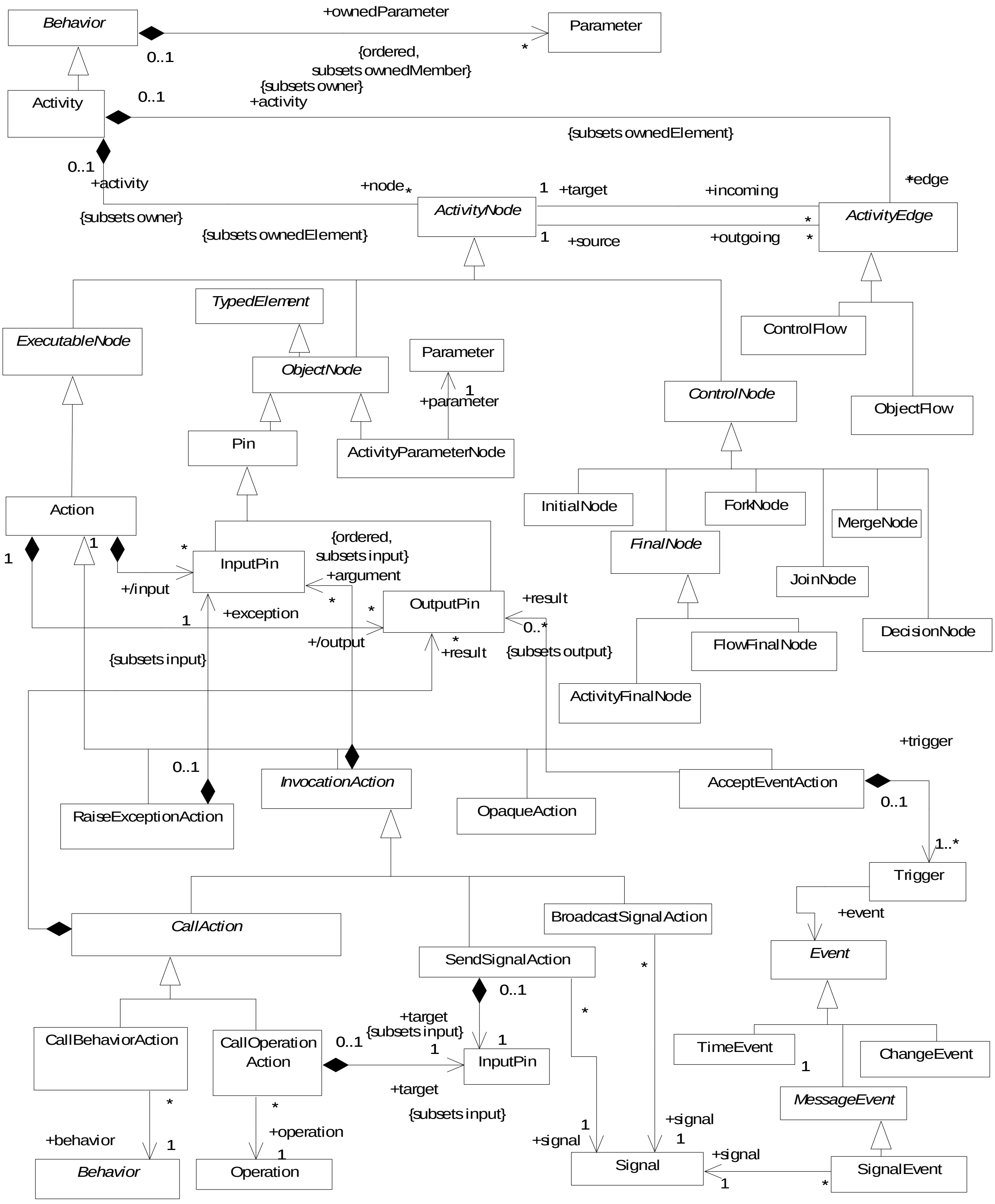


Figure 5.9. Subset of the UML2.0 Activity and Action concepts we identified as a basis of UML4SPM

4. UML4SPM Notations

The graphical representation of a UML4SPM *Software Activity* is given in figure 5.10. As we can notice, it differs slightly from the one proposed by the UML2.0

standard. This is because it owns new properties and associations specific to software process modeling that we newly defined.

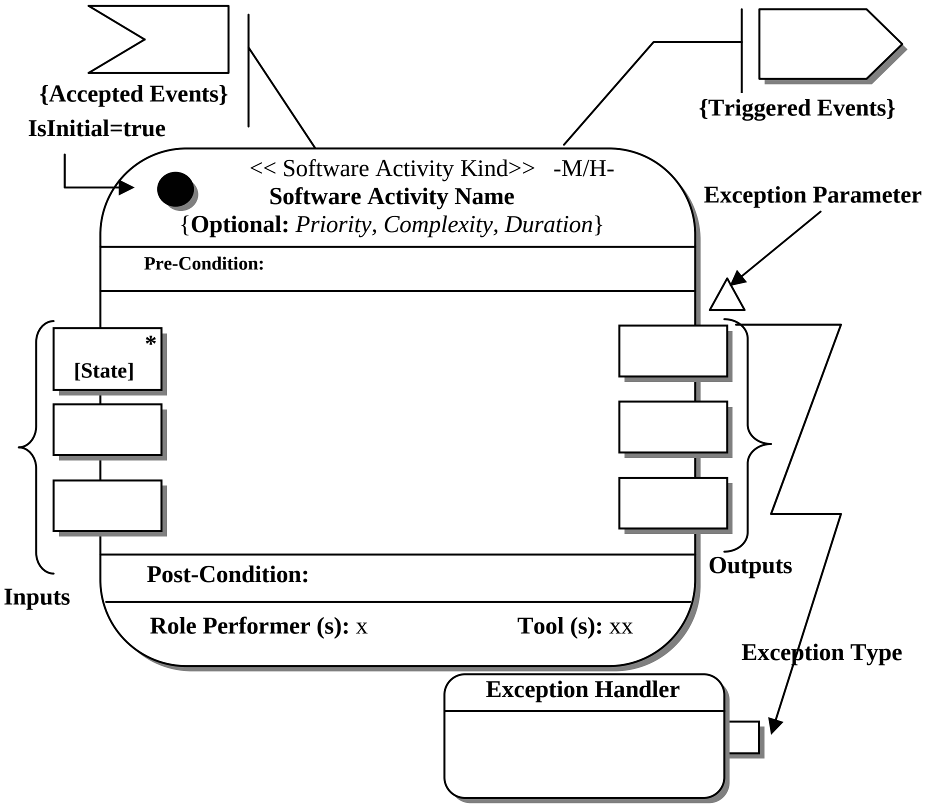


Figure 5.10. UML4SPM Software Activity Notation

Precision was a major requirement for this notation. At a glance, the *Agent* or the developer can know the name of the activity, its input and output parameters, its priority in the process, its duration, activity post and pre conditions, its assigned agents, tools required for performing the activity, accepted and triggered events.

We also allow the possibility to express the multiplicity of *Software Activity* parameters and their states. A star sign (*) on the top-right corner of a *Software Activity* parameter means that while calling the activity, the parameter may be omitted. This is a very powerful feature since we can have the same activity that might be called by different activities, many times, with different parameters.

A state machine-like initial blob may be placed on the top-left corner of the *Software Activity* to distinguish between the initial activity and the others. To indicate that the *Software Activity* is totally machine-executable an "M" letter is placed on top of the activity name. Otherwise, an "H" letter is to indicate that human expertise is required.

As we saw in the UML4SPM metamodel description (cf. Section 3), there is a possibility to model user-defined and method-specific kinds of *Software Activities*. This is done through the *Software Activity Kind* element. The *Kind* of the *Software Activity*, if any, is given between double occurrences of the less than symbol "<<" and greater than symbol ">>" as it is done for UML Profile Stereotypes. Examples of *Software Activity Kinds* would be <<Phase>>, <<Process>>, <<Iteration>>, etc.

A *Software Activity* may be protected by an *Exception Handler* in case an exception occurs. The exception parameter and the exception type may be explicitly indicated on the activity representation. Of course, the presentation of all these pieces of information is optional and the modeller has to choose the most relevant ones for process description.

The complete presentation of the notation of *Software Activity* constituents, which represent a subset of UML2.0 *Activity* elements, is given in Appendix A. Some notations are modified or enriched with symbols in order to take into account some element's important features (properties) for process modeling purposes (e.g. to indicate that the *CallBehaviorAction* is synchronous or not). Additionally, for UML2.0 *Activity* and *Action* concepts, which do not have a notation, we propose one. Semantics of these concepts are given in more details in [OMG 07b]. Some of them are addressed in the next chapter through the UML4SPM evaluation and in Chapter 7 and 8 for UML4SPM process model executions.

5. Conclusion

We started this chapter by giving our main design goals for UML4SPM which are: the raise in *Abstraction*; the use of a *Standard* and *Well-Known* formalism; and finally, *Executability*. The aim of this chapter was to present the UML4SPM metamodel and to give an insight of some of the language features that might help us in reaching these goals.

As a basis of our software process modeling language, we opted for reusing some UML2.0 *Activity* and *Action* concepts instead of starting from scratch. After the identification of these concepts for their appropriateness for process modeling, they are extended or reused by the metamodel we defined, which was validated in [Bendraou 05a]. While UML2.0 *Activity* elements provide mechanisms and concepts for control flows, triggering of events, exception handling, constraints, etc, our metamodel defines the set of concepts with semantics proper to software process modeling. UML2.0 is a standard and well-known modeling language providing high-level abstractions, which would help us in satisfying the "*Abstraction*" and "*Standard-based*" design goals. The potential of UML4SPM in terms of expressiveness and understandability are demonstrated in the next chapter through a process example and the language is evaluated according to the criteria we defined in the previous chapter.

Regarding *Executability*, we presented in this chapter the capability to express actions with executable semantics within UML4SPM process models. This is made possible by reusing some of the action elements proposed by UML2.0. At this effect, we identified the set of actions suitable for process modeling and how they can be reused to coordinate process activity executions. To reach the *Executability* requirement, we will explore different solutions to execute these actions. These solutions are presented in Chapter 7 and Chapter 8.

Finally, we presented the UML4SPM notation, which is mainly based on UML2.0 *Activity* notations. Some modifications were introduced in order to take into account some features proper to software process modeling but also to increase understandability. For UML2.0 *Activity* elements and *Actions* for which no notation is proposed by the standard, we proposed one.

Chapter 6

UML4SPM Language Evaluation

1. Introduction

In the previous chapter we introduced the UML4SPM metamodel and the notation of its elements. The aim of this chapter is to evaluate the expressiveness of the UML4SPM Process Modeling Language. The evaluation is done according to the SPML requirements we defined in Chapter 4 of this document. The different capabilities of the language are presented except *Executability* which is addressed in the next chapters (i.e., Chapter 7 and 8). This evaluation is presented in the Section 2.

In Section 3, we use UML4SPM for modeling a part of the well-known ISPW-6 software process example, a benchmark for comparing and evaluating software process modeling approaches. A discussion on the outcomes of this evaluation is given at the end of the section.

Finally, some observations on the result we have through the evaluation of UML4SPM conclude this chapter.

2. Evaluation of UML4SPM according to SPML Requirements

After having introduced the UML4SPM metamodel in the previous chapter, in the following, we go through the requirements we defined in Chapter 4, in order to evaluate our proposition. We start by the semantic richness requirement.

2.1. Semantic Richness

As we saw in Chapter 4, this requirement covers many aspects. Herein, we detail each of them in the context of UML4SPM.

Process Elements

The notion of ***Activity*** is given through the *Software Activity* metaclass. In UML4SPM, a *Software Activity* represents any effort or piece of work to be performed during the software development process. It has a kind specified by the *Software Activity Kind* element. Thus, it is rendered possible to define any user-defined or methodology-specific kind of activities. A *Software Activity* can be a "Process", a "Phase", an "Iteration", a "Sprint", etc.

The notion of ***Role*** is offered by the *Responsible Role* metaclass. As for software activities, it is possible to define process and domain-specific roles thanks to the *Responsible Role Kind*. A *Responsible Role* may be in charge of one or more software activities but also of *WorkProducts*. UML4SPM *WorkProduct* element is the equivalent of ***Artifact*** and may have specific kinds such as "Document", "Model", "Code", etc.

A *Responsible Role* can be undertaken by one or more *Role Performers*. The *Role Performer* is the abstraction of the elements suitable for performing a role. It regroups the *Team*, *Agent* (which represents the notion of **Human**) and ***Tool*** metaclasses. A *Team* may be composed by *Agents* or by other *Teams*.

The set of concepts used in UML4SPM are not limited to basic process elements [Dowson 91], [Humphrey 92], [Conradi 95], but also includes, the notion of *Guidance, Time Limit*, *Constraints*, *Actions*, etc (cf. Chapter 5 Section 3).

Coordination and Sequencing of Activities and Actions

In UML4SPM, there are two levels of sequencing. *Actions* sequencing and *Software Activities* sequencing.

Sequencing and Coordination of Action

Regarding *Actions* sequencing, a *Control Flow* between an action "A" and an action "B" is to indicate that "B" starts when "A" is finished. This is the means to express the basic *finish-start* precedence between *Actions*. Actions being atomic units of work, the *finish-start* precedence is the only precedence relationship that we can have between actions.

Control Flows that are more complicated can be expressed by combining the use of *Control Flow* and specializations of *Control Nodes* (figure. 6.2.). A *Control Flow* combined with a *Fork* node allows the specification of parallel action executions. A *Join* node is used to synchronize multiple flows that is, the action after the *Join* node will not start until all actions corresponding to incoming flows of the *Join* node terminate. One advanced property of the *Join* node is the "JoinSpec" property, a specification giving conditions under which the *Join* will succeed. An example using a *Join Node* is given in figure 6.3. Unlike the *Join* node, in the *Merge* node, the completion of one among multiple alternate actions would be sufficient to start the action after the node.

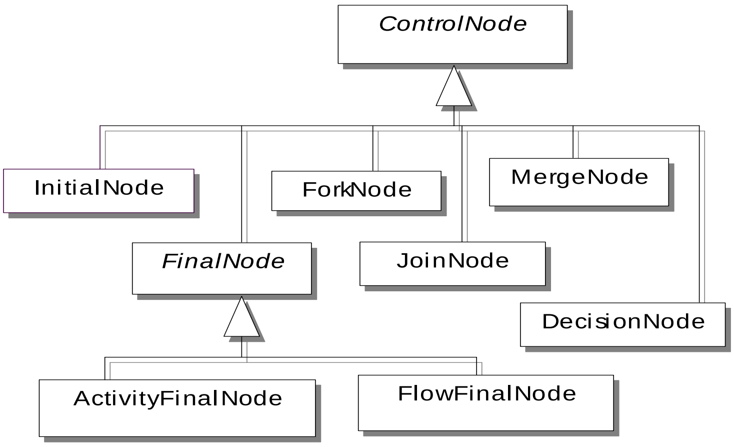


Figure 6.2. Control Nodes

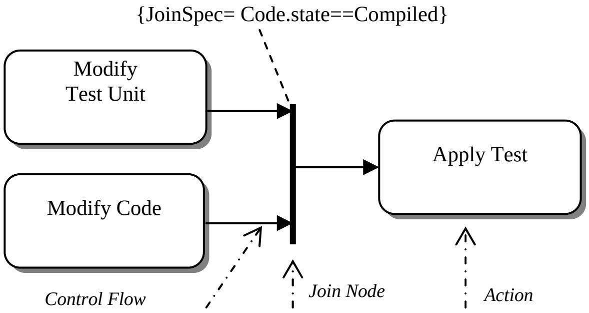


Figure 6.3. An example using a Join Node

In the absence of an explicit *Control Flow*, an *Object Flow* can be used instead. An *Object Flow* between an action "A" and an action "B" is to indicate that the data or the object produced by "A" is to be consumed by "B". As *Control Flows, Object Flows* can be combined with *Control Nodes* for specifying flows that are more complex.

Sequencing and Coordination of Activities

For *Software Activity* coordination within UML4SPM, we take advantage of *Events* and *Invocation Actions* of UML2.0. In the past, experiences with first-generation PMLs using events have demonstrated their effectiveness [Cohen 88], [Sutton 95b] and [Dami 98].

UML2.0 proposes three kinds of events: *Message Event*, *Time Event* and *Change Event*. *Message Event* specifies the receipt by an object (in our case, an instance of *Software Activity*) of either an *Operation Call* or a *Signal*. When a *Signal* is sent for instance by an activity "A" to activity "B", its reception is handled by an *AcceptEventAction*, which may trigger a behavior specified within "B".

The *Change Event* models a change in the system configuration that makes a condition true. It allows *Software Activities* to react instantaneously to *WorkProduct* state changes or to the completion or starting of other *Software Activities*.

Message Event and *Change Event* represent a very powerful mechanism to activate the execution of an activity without using a *Control* or an *Object Flow*, contrarily to most PMLs where the activation of an activity had to be specified explicitly through control flows. They are also the means to coordinate and to synchronize between executions of different *Software Activities*. Combined with *Control Nodes*, *Message Event* and *Change Even* allow the modeling of all kinds of precedence relationships between activities (i.e., *start-start*, *start-finish*, *finish-start*, and *finish-finish*).

The last kind of event is the *Time Event*. It specifies a point in time. At the specified time, the event occurs. This is very useful for instance to model a monitoring progress activity within a software development process. When an activity deadline event occurs, the project manager, depending on the activity progression, will decide either to extend its deadline or to pass by to another one.

Another means to initiate a *Software Activity* is by using the *CallBehaviorAction*, a specialization of UML2.0 *Invocation Actions*. It allows the activation of a *Software Activity* execution from another one. Thus, in absence of Events, this mechanism can be used for activity sequencing. One important advantage is that the called *Software Activity* can start without requiring that the caller one terminates. The call may be synchronous or asynchronous which determines if the caller activity has to wait for the execution result of the called activity or can continue executing. Combined with *Control Nodes*, the *CallBehaviorAction* can be used to synchronize between executions of different activities. For example, the parallelization of *Software Activity* invocations can be expressed by combining the *CallBehaviorAction* with a *Fork Node*.

Another significant advantage is that parameters can be added to the *Software Activity* call. Parameters may be typed or untyped and have a multiplicity which adds more flexibility. Thus, through this mechanism, *WorkProducts* (being *Classifiers*) can be exchanged or transferred across *Software Activities* as it is done in common programming languages where parameters are passed to operations through operation calls. *Software Activity* parameters are represented via the *Activity Parameter Node* concept. Examples of the use of that latter and the *CallBehaviorAction* will be illustrated in Section 4.

Workflow Patterns

Additionally to control flows and precedence relationships presented previously, there is what we call *Workflow Patterns*. Workflow Patterns represent some recurrent business situations and problems that one may have to describe within process models. The Workflow Patterns initiative is a joint effort of Eindhoven University of Technology (led by Professor Wil van der Aalst) and Queensland University of Technology (led by Associate Professor Arthur ter Hofstede) which started in 1999 [WfP].

The aim of this initiative was to provide a conceptual basis for process technology. In particular, the research provides a thorough examination of the various perspectives (control flow, data, resource, and exception handling) that need to be supported by a workflow language or a business process modelling language.

The results can be used for examining the suitability of a particular process language or workflow system for a particular project, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular process-aware information system. They can also be used as a basis for language and tool development.

This initiative distinguishes between four kinds of Workflow Pattern perspectives. The *Control-Flow* perspective, which captures aspects related to control-flow dependencies between various tasks (e.g. parallelism, choice, synchronization etc). Originally twenty patterns were proposed for this perspective, but in the latest iteration this has grown to over forty patterns. The *Control Flow* perspective is of particular interest for our purpose. The *Data* perspective deals with the passing of information , scoping of variables, etc, while the resource perspective deals with resource to task allocation, delegation, etc. Finally the patterns for the *Exception Handling* perspective deal with the various causes of exceptions and the various actions that need to be taken as a result of exceptions occurring.

Workflow Patterns are described through: conditions that should hold for the pattern to be applicable; examples of business situations; problems, typically semantic problems, of realization in current languages; and implementation solutions [Van Der Aalst 03a].

The Workflow Patterns initiative evaluated many propositions and standards for process modeling covering both workflow and business process domains. Examples of such standards are (BPMN, XPDL, BPEL, UML, etc.).

The evaluation of UML2.0 dealt with the capacity of UML2.0 *Activity* diagrams to support the set of patterns defined by the group. UML2.0 Activity diagrams proved their expressiveness and ability to represent most of the significant *Control Flow* patterns. More than thirty patterns of the forty control flow patterns inventoried by the group were satisfied. The list of these control flow patterns is presented in table 6.1. If a standard directly supports the pattern through one of its constructs, it is rated +. If the pattern is not directly supported, it is rated +/-. Any solution which results in spaghetti diagrams or coding, is considered as giving no direct support and is rated -. Note that a pattern is only supported directly if there is a feature provided by the language which supports the construct without resorting to any of solutions mentioned in the implementation part of the pattern.

More details on this evaluation are given in [Wohed 04] [Russel 06]. Comparison between UML2.0 and other standards for business process modeling such as BPEL, XPDL, etc and the description of each pattern is given in [WfP].

Additionally to its high abstraction level and the fact that it is standard and wild-spread, UML2.0 also provides some powerful mechanisms to deal with most complicated control flows. These results reinforce our choices in using the UML2.0 *Activity* concepts as a basis of UML4SPM.

	UML2.0
Control-Flow Patterns	
Sequence	+
Parallel Split	+
Synchronization	+
Exclusive Choice	+
Simple Merge	+
Multi-Choice	+
Structured Synchronizing Merge	-
Multi-Merge	+
Structured Discriminator	+/-
Arbitrary Cycles	+
Implicit Termination	+
Multiple Instances without Synchronization	+
Multiple Instances with a Priori Design-Time Knowledge	+
Multiple Instances with a Priori Run-Time Knowledge	+
Multiple Instances without a Priori Run-Time Knowledge	-
Deferred Choice	+
Interleaved Parallel Routing	-
Milestone	-
Cancel Activity	+
Cancel Case	+
Structured Loop	+
Recursion	-
Transient Trigger	+
Persistent Trigger	+
Cancel Region	+
Cancel Multiple Instance Activity	+
Complete Multiple Instance Activity	-

Blocking Discriminator	+/-
Canceling Discriminator	+
Structured N-out-of-M Join	+/-
Blocking N-out-of-M Join	+/-
Canceling N-out-of-M Join	+
Generalized AND-Join	-
Static Partial Join for Multiple Instances	-
Canceling Partial Join for Multiple Instances	-
Dynamic Partial Join for Multiple Instances	-
Acyclic Synchronizing Merge	+/-
General Synchronizing Merge	-
Critical Section	-
Interleaved Routing	-
Thread Merge	+
Thread Split	+
Explicit Termination	+

Table 6.1. Control-Flow Patterns supported by UML2.0 Activity Diagrams

Exception Handling

Like *Events*, exception handling in UML4SPM provides *Software Activities* with a reactive control flow. Thus, in case of exceptions, this mechanism will redirect the control flow to a predefined behavior. That latter is also described in terms of a *Software Activity* containing *Actions* or may be implementation-specific. Exception handling is ensured thanks to the UML2.0 *RaiseExceptionAction* and *ExceptionHandler* elements. In case of an unexpected situation during the *Software Activity* execution, a *RaiseExceptionAction* occurs indicating the exception type. All flows within the *Software Activity* are then stopped and the appropriate handler is assigned. In the literature, exception handling is not well addressed by PML and only few ones address it [Cass 00].

Advanced Constructs:

To add more flexibility, *Decision Nodes* are provided. They offer to the process modeler the ability to express conditional branches with *Guards* under which the control flow will be directed if it is evaluated at true. To describe iterations, the modeler can use the *Loop Node* element and for more structured decisions, she/he can use *Conditional Nodes*. These two last elements are very similar to common loop and conditional constructs in usual programming languages.

A weakness of most PMLs is the absence of elements that model the storage and retrieval of *WorkProducts* used during the *Software Activity* performing. Among exceptions, the HI-PLAN process modeling language. It proposes the concept of *Deliverables Store* [Hyungwon 96], a physical storage of related artifacts that can be used or produced by an activity.

In UML4SPM, we decided to reuse an UML2.0 construct with a similar semantic and which fulfills this role. A *DataStoreNode* in UML2.0 models a kind of a buffer for no transient information (i.e., persistent). It keeps all tokens that enter to it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any token in the *DataStoreNode* containing that object. In

UML4SPM, we take advantage of this element while replacing the notion of token with *WorkProduct*. *WorkProducts* management versioning is ensured thanks to the "version" property we defined for each *WorkProduct*.

2.2. Understandability

Understandability is a very crucial requirement. It is what can induce people to use one PML instead of another. This is what has strongly influenced us at investigating the reusability of UML2.0 for software process modeling [Bendraou 05].

UML4SPM reuses a set of UML2.0 elements and notations. This is considered as a serious advantage given that UML has attractive features. It is standard, graphical, intuitive, and easy to understand. A wide community of software developers is already familiar with UML and variety of tools and training supports are proposed. UML being so popular and widely used, UML4SPM has an important competitive advantage compared to any specialized PML.

For documenting and specifying software processes, we decided to exploit UML2.0 diagrams while restricting those to be used as in SPEM1.1 [OMG 02].

In UML4SPM, the UML2.0 *Activity* diagram is used to model the sequencing of *Software Activities* and *WorkProducts* exchange between *Actions*. The *State Machines* diagram is used to model the allowable states and operations of *Software Activities*, *WorkProducts* or *Agents*. For the specification of operation calls between *Software Activities*, process modelers can use a *Sequence* diagram. Finally, *Class* diagrams are used to show the relationships between different process model elements (i.e., inheritance, dependency, associations). An example of its use would be for instance to represent the "nestedArtifacts" and the "impacts" associations between different *WorkProducts*. So in one sight, we can see which *WorkProduct* is part of other *WorkProducts* and those it may impact if it has to be modified.

2.3. Precision

The building block of UML4SPM process models is the *Software Activity* element. A *Software Activity* may be composed of other *Software Activities* and has a *Kind*. Example of *Software Activity* kinds are a "Process", a "Sprint", a "Task", a "Phase", an "Iteration", an "Activity", etc. Thus, it is possible to represent any user-defined or methodology-specific hierarchy of processes. For instance, in the RUP process [Kruchten 03], it is possible to define different level of hierarchy. A Process may be composed of Disciplines. A Discipline may be composed of Phases. A Phase may be composed of Iterations. Iteration may be composed of Activities and so on. With UML4SPM, it is possible to represent such hierarchy concepts by combining the *Software Activity* and the *Software Activity Kind* elements.

Software Activities can be also described in terms of *Actions* with computational semantics. This is due to the fact that the UML4SPM *Software Activity* element extends the UML 2.0 *Activity* one.

For modeling process's activities, UML4SPM offers two levels of abstractions. The *Process* view which aims at giving an abstract description of all process's *Software Activities*. The *Activity* view which aims at modeling every *Action* to be performed during the *Software Activity* with its constraints, events, inputs and outputs in terms of *WorkProducts* with their actual state and their type.

In Section 4, we give a description of the ISPW-6 process example while using both views. Also, we can notice that the fact of using UML2.0 *Actions* within UML4SPM not only brings precision, but it also provides process models with the degree of details required to execute them. This is how precision relates to executability which presented in the next subsection.

2.4. Executability

The intent of UML2.0 *Activity* constructs has changed radically from UML1.x. *Activities* are not only suitable to model processes; they also have some features necessary to support the automation of these processes [Hausmann 05].

Besides, UML2.0 offers four *Action* packages (*BasicActions, IntermediateActions, Structured Actions and CompleteActions*) in order to express the semantic of most executable actions that we can find in common programming languages. Thus, the specification of software process models with operational semantics is rendered possible. This facility makes it possible to automate the mapping of UML4SPM process models towards programming languages or workflow and business process execution formalisms in order to execute them. In the previous chapter (c.f. Section 3.2.3.) we identified the set of *Actions* suitable for process executions. In chapter 7, we will explore the possibility to map UML4SPM process models towards a well-know formalism for business process execution called BPEL (Business Process Execution Language) [WSBEPL 07].

Some efforts were also done as an attempt to formalize UML2.0 *Activities* [Vitolins 05] [Sarstedt 06] and UML virtual machines and simulators are already under study in some research projects [UML 06] [STL 06] [MODELWARE].

Furthermore, the OMG issued a new RFP (Request For Proposal) named: *Executable UML Foundation* [OMG 05c]. The objective of this RFP is the definition of a computationally complete and compact subset of UML 2.0 to be known as "Executable UML Foundation", along with a full specification of the execution semantics of this subset. "Computationally complete" means that the subset shall be sufficiently expressive to allow definition of models that can be executed on a computer either through interpretation or as equivalent computer programs generated from the models through some kind of automated transformations. This initiative is also explored for the purpose of UML4SPM process model executions. In Chapter 8, an introduction to the approach as well as the implementation we propose are presented.

All these initiatives we just enumerated and the potential that UML2.0 *Activity* and *Actions* provide in terms of executability comfort us in our decision of reusing UML2.0 as a basis of UML4SPM.

2.5. Modularization

Considering that a UML4SPM *Software Activity* can define an internally consistent Process (respectively, an internally consistent Phase, Iteration, Sprint, Activity, etc. depending on the *Software Activity Kind* value), *Software Activities* are then considered as modularization units. To combine, to coordinate or to compose a new *Software Activity* from (between) other *Software Activities* we take advantage of the flexibility offered by the *CallBehaviorAction*.

The *CallBehaviorAction* allows to *Activities* to be interconnected in a practical way. The advantage of this construct is that behaviors are invoked as it is done for methods in classical programming languages. Making this way, modelers don't have to carry out the *unification* of *Software Activities* inputs and outputs (i.e., make names of a Software Activity's outputs identical with another Software Activity's inputs).

In Java for instance, parameters of a method call can be named differently in the operation signature. *CallBehaviorAction* being a *CallAction*, casting of parameters is done implicitly when activities are invoked thanks to the abstraction given by *InputPins* and *OutputPins* concepts.

In figure 6.4., we give an example in order to demonstrate how *CallBehaviorAction* can be used to allow *Software Activity* compositions. The example introduces two *Software Activities*. Shaded boxes of the figure represent the "Class Diagram Realization" activity. It is composed of a set of actions linked through control flows (not represented in the figure for readability sake). The aim of these actions is to guide the developer through the construction of a UML Class Diagram. One output of these actions is the Class Diagram *WorkProduct* (CD in the figure).

This output is used as an input argument when calling the "Class Diagram-To-RDB Transformation" *Software Activity* which is represented in lighted boxes in the figure. The action used to connect between both activities is the *CallBehaviorAction*. The two activities are interconnected thanks to *ActivityParameterNode* and no unification procedure is needed.

Then, *Software Activity* compositions are realized just by adding a *CallBehaviorAction*. A new Process definition can be established simply by defining an orchestration of *CallBehaviorActions* aiming at invoking the different *Software Activities* composing the process. They can even be specified at execution-time. This offers more flexibility and spares many efforts to process modelers.

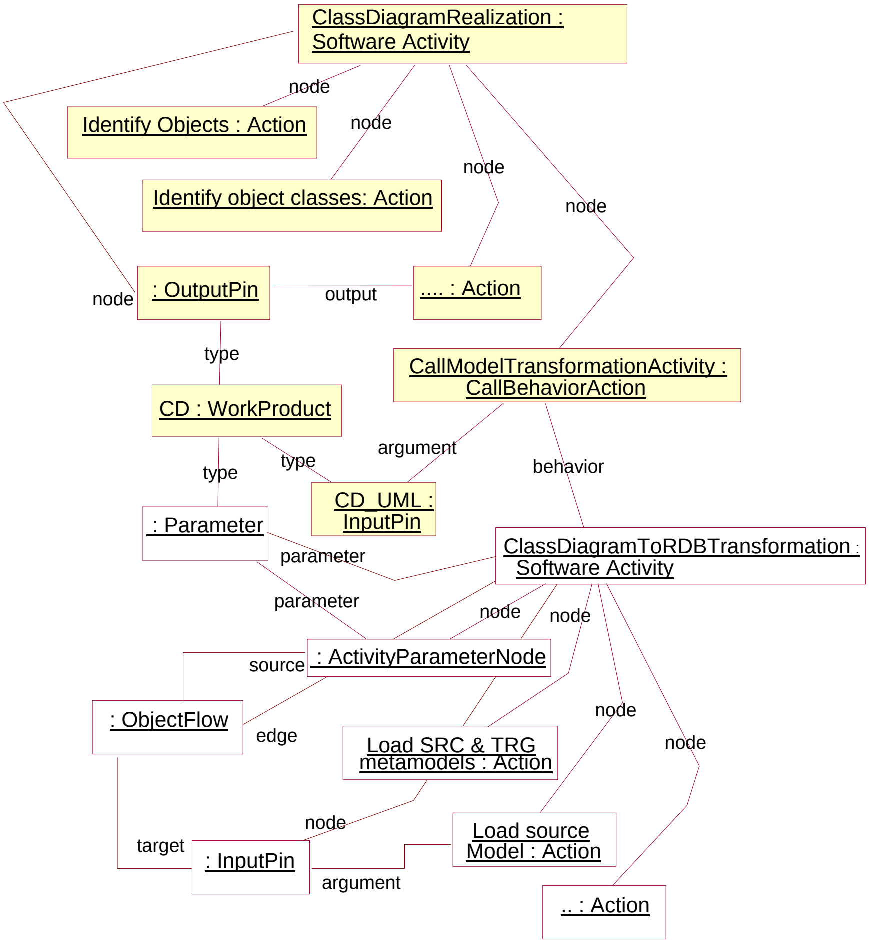

ClassDiagramRealization : Software Activity
node
Identify Objects : Action
node
Identify object classes: Action
node
node
node
: OutputPin
output
.... : Action
type
CallModelTransformationActivity : CallBehaviorAction
CD : WorkProduct
argument
type
type
behavior
CD_UML : InputPin
: Parameter
parameter
ClassDiagramToRDBTransformation : Software Activity
parameter
node
node
source
: ActivityParameterNode
: ObjectFlow
edge
node
node
Load SRC & TRG metamodels : Action
node
target
: InputPin
Load source Model : Action
argument
.. : Action

Figure 6.4. Software Activity interconnections thanks to the CallBehaviorAction.

3. Evaluation of UML4SPM with the ISPW-6 Software Process Example

In order to evaluate the expressiveness of our language, we found it interesting to use the ISPW-6 process example as basis of our evaluation. In the following, we start by introducing the software process example as well as the motivations that led to this proposition. Then, UML4SPM is used to model some of the process example tasks.

3.1. ISPW-6 Software Process Example

In order to compare and to understand the various software process modeling approaches, a working group, in conjunction with the sixth International Software Process Workshop (ISPW-6), has developed a benchmark consisting in a software process modeling example problem [Kellner 91b]. The original ISPW-6 Software Process Example was carefully designed to incorporate important process aspects and issues. The aim behind is to aid in the evaluation and deduction of the relative strengths and weaknesses of the process modeling approach under examination. Different modeling approaches have been applied to this common process example [Kellner

91a]. As a result, they have extended and enhanced their approaches in response to working through it.

However, it should be recognized that a determination of strengths and weaknesses must be based upon some set of goals and objectives which a given approach is intended to achieve. For example, direct executability is of paramount importance for an approach whose major goal is to provide automated support for process enactment; on the other hand, it may be relatively inconsequential for an approach focused upon facilitating human understanding and communication regarding the process.

In order to facilitate understanding and comparisons, this problem has been painstakingly designed to contain a large number of different types of process issues seen in real-world software processes. This provides an opportunity to demonstrate the capability to model over a dozen different categories of process issues.

The example problem consists of a core problem and several optional extensions. Solution of the core problem is required, in order to provide a common ground for beginning to understand different modeling approaches. The optional extensions provide additional opportunities to demonstrate the capabilities of different approaches, and are much more open-ended.

The Core Problem

The core problem focuses on the designing, coding, unit testing and management of a rather localized change to a software system. This is prompted by a change in requirements and can be thought of as occurring either late in the development phase or during the support phase of the life cycle.

The problem begins with the project manager scheduling the changes and assigning the work to appropriate staff. The example ends when the new version of the code has successfully passed the new unit tests.

The component activities (steps) of this process example are:

1. Schedule and Assign Tasks;
2. Modify Design;
3. Review Design;
4. Modify Code;
5. Modify Test Plans;
6. Modify Unit Test Package;
7. Test unit;
8. Monitor Progress.

The details of what has to be performed within each activity, their sequencing, their inputs, outputs, roles, and constraints are given in [Kellner 91b].

Optional Extensions

Optional extensions to the core problem aim to provide a variety of issues for showcasing capabilities of a given modeling approach that may not be demonstrated by the core problem. Each extension is built upon the core problem, and they can be considered independently of each other. They are rather open-ended, leaving considerable freedom to orient them as desired.

Examples of these extensions are to provide the means to express automatic tool invocations, to differentiate between different product versions during code modifications, improve communication between agents, express modeler's choices and process change.

3.2. Modeling the ISPW-6 Software Process Example with UML4SPM

In this chapter we are not going to include all ISPW-6 process tasks we modeled using UML4SPM. They are given in Appendix A of this document. Hereunder, we illustrate the use of UML4SPM notation and we highlight some important features of the language through the modeling of the *Modify Design* and *Review Design* tasks. In the following, we present each task. The way these tasks relate to other process's tasks can be found in [Kellner 91b].

4.2.1. Modify Design

Description

This step involves the modification of the design for the code unit affected by the requirements change. It is a highly creative task. The modified design will be reviewed, and ultimately implemented in code. This step may also modify the design based upon feedback from the design review.

Inputs

1. Current design (from software design document file) (hand carried)
2. Design review feedback (from design review) (hand carried)

Outputs

1. Modified design (to review design, modify code, modify unit test package) (hand carried)

Responsibility

This step is carried out by the assigned design engineer.

Constraints

1. This step can begin as soon as the task has been assigned by the project manager.
2. Subsequent iterations can begin as soon as the design review is completed (when the design is not approved).
3. This step ends when its output has been provided.

UML4SPM Notation

Figure 6.5 represents the *Modify Design* task modeled thanks to UML4SPM.

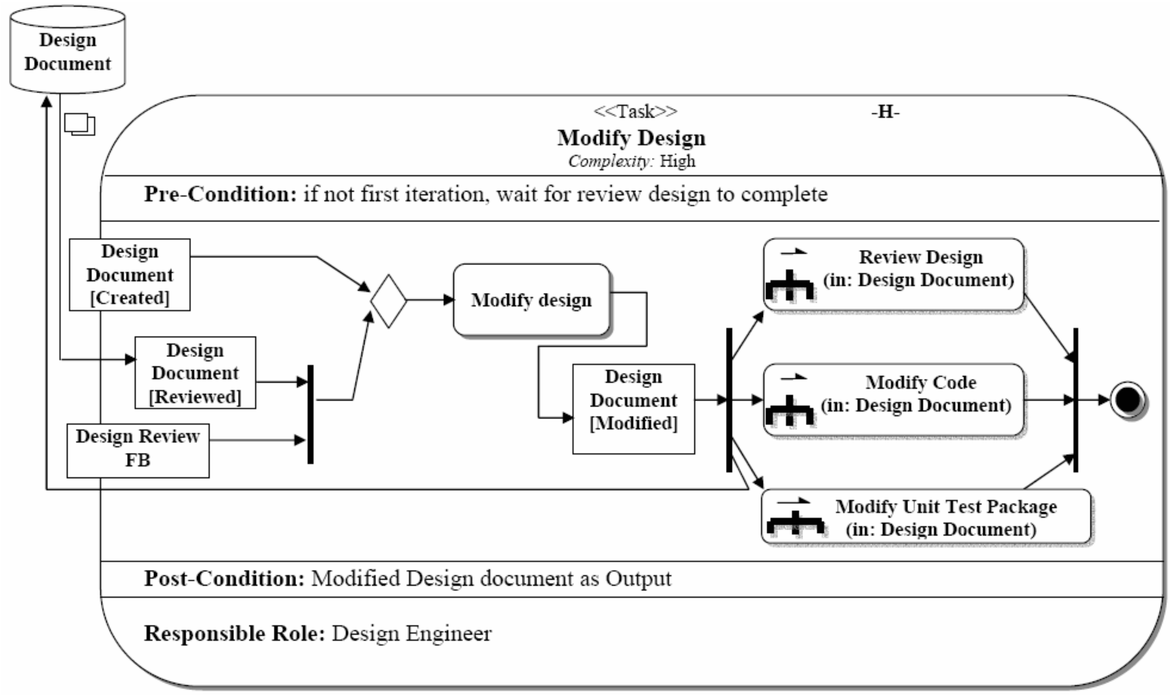

Design Document
<<Task>>
-H-
Modify Design
Complexity: High
Pre-Condition: if not first iteration, wait for review design to complete
Design Document [Created]
Design Document [Reviewed]
Design Review FB
Modify design
Design Document [Modified]
Review Design (in: Design Document)
Modify Code (in: Design Document)
Modify Unit Test Package (in: Design Document)
Post-Condition: Modified Design document as Output
Responsible Role: Design Engineer

Figure 6.5. The Modify Design Task

Looking at the figure, we can notice many aspects. First, the use of a *Merge Node* in order to accept one flow among the two flows incoming to the node. The "Modify Design" action will not start unless one of these flows is activated. The first flow is an *Object Flow* and comes from the "Design Document" *ActivityParameterNode*. This flow is activated when the activity is called for the first time. The second flow, also an *Object Flow*, comes from the "Design Review Feed Back" *ActivityParameterNode* and may be activated when calling the activity from the *Review Design* task. Another use of control nodes is illustrated with the *Join Node* which is used for synchronizing the reception of the "Design Document" (with state Reviewed) and the "Design Review FB" document.

The second aspect relates to the use of the *CallBehaviorAction* in order to activate the execution of another activity. To express parallel activity calls, a *Fork Node* is combined with three *CallBehaviorActions* (*Review Design, Modify Code and Modify Unit Test Package* activity calls). A half arrow on the *CallBehaviorAction* is to indicate that the call is asynchronous. We also add the possibility to express the parameter of the call as well as its kind (i.e., in, out, inout).

Other important aspects are the *WorkProduct* persistency and management. At this aim we use the *DataStoreNode* element represented in the figure by a cylinder. An arrow and a double rectangle from the *DataStoreNode* mean that a copy of the *WorkProduct* is offered to the action. An incoming arrow to the *DataStoreNode* means that *WorkProduct* version entering the node will replace the existing one. *WorkProducts* are represented with their actual states to avoid any confusion however it is not mandatory.

Finally, we can notice that the *Kind* of the *Software Activity* is set to <<Task>> according to the context of the ISPW-6 process example, that it has to be carried out by a human (the "H" symbol at the top-right corner) and its complexity is fixed at "high".

This would imply that the activity need to be affected with very skilled people and may need more attention during the process execution.

4.2.2. Review Design

Description

This step involves the formal review of the modified design. It is conducted by a team including the design engineer who produced the design modifications. There are three alternative outcomes of the review:

1. Unconditional approval -- The design is totally approved; the approved modified design is incorporated into the software design document.
2. Minor changes recommended -- Minor changes to the design are required and feedback is provided to the designer. The re-review is expected to be perfunctory.
3. Major changes recommended -- Major changes to the design are required and feedback is provided to the designer.

At the conclusion of the review, the project manager is notified of the outcome. Due to the fact that formal design reviews are a relatively new step in this organization's process, they are recording certain measurements to help evaluate its impact. In particular, the number of defects identified, and the aggregate effort of the review team in preparing for and conducting the review are reported to the project manager.

Inputs

1. Modified design (from modify design) (hand carried)

Outputs

1. Design review feedback (to modify design) (hand carried)
2. Approved modified design (to software design document file) (hand carried)
3. Outcome notification, number of defects identified, aggregate effort (to monitor progress) (e-mail)

Responsibility

This step is carried out by the design review team assigned by the project manager. This team includes the design engineer, QA engineer, and two other software engineers.

Constraints

1. This step will be carried out when it is scheduled to occur, provided that the modified design is available at that time. (Note that in cases of delay, the monitor progress step will reschedule the review to a later date.)
2. This step ends when its outputs have been provided; assume that all outputs are produced simultaneously.

UML4SPM Notation

Figure 6.6 represents the Review Design task modeled using UML4SPM

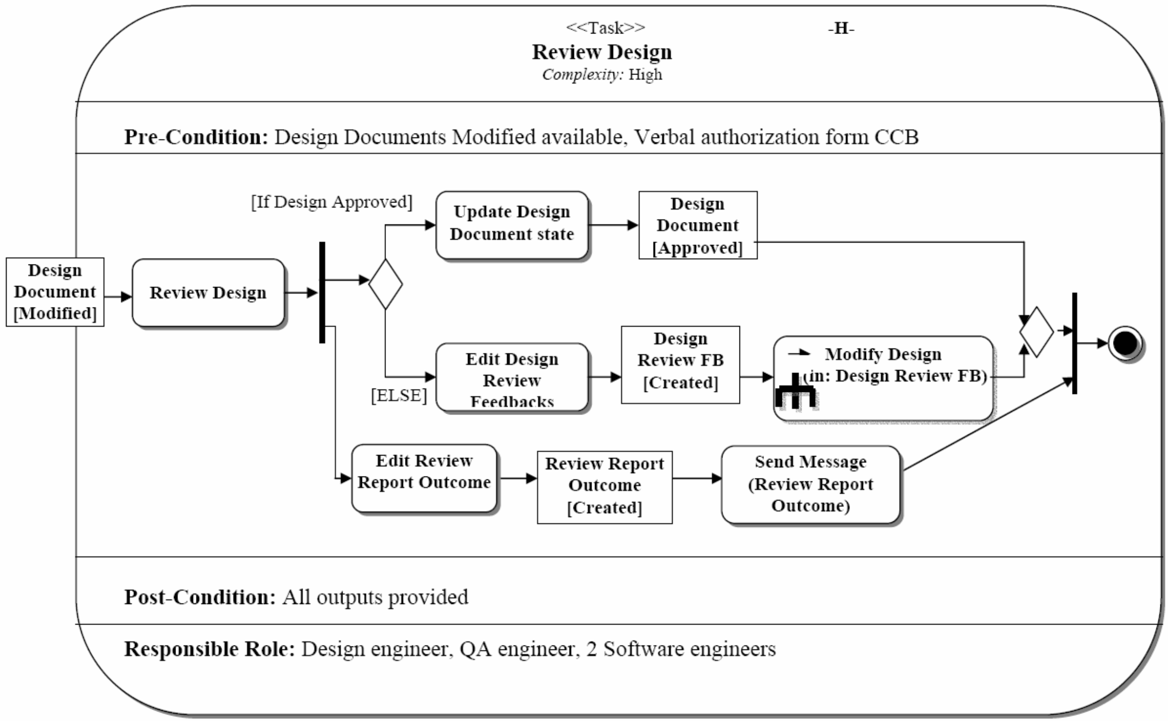

<<Task>>
-H-
Review Design
Complexity: High
Pre-Condition: Design Documents Modified available, Verbal authorization form CCB
[If Design Approved]
Update Design Document state
Design Document [Approved]
Design Document [Modified]
Review Design
Edit Design Review Feedbacks
[ELSE]
Design Review FB [Created]
Modify Design (in: Design Review FB)
Edit Review Report Outcome
Review Report Outcome [Created]
Send Message (Review Report Outcome)
Post-Condition: All outputs provided
Responsible Role: Design engineer, QA engineer, 2 Software engineers

Figure 6.6. Review Design Task

The *Review Design* is activated from the *Modify Design* task. This is done thanks to an asynchronous *CallBehaviorAction* that transmit the modified *Design Document* to the *Review Design* task. Once the *ActivityParameterNode* triggered by the reception of the document, the "Review Design" action can start. The absence of the multiplicity tag (i.e. "*") means that the parameter is mandatory for executing the activity.

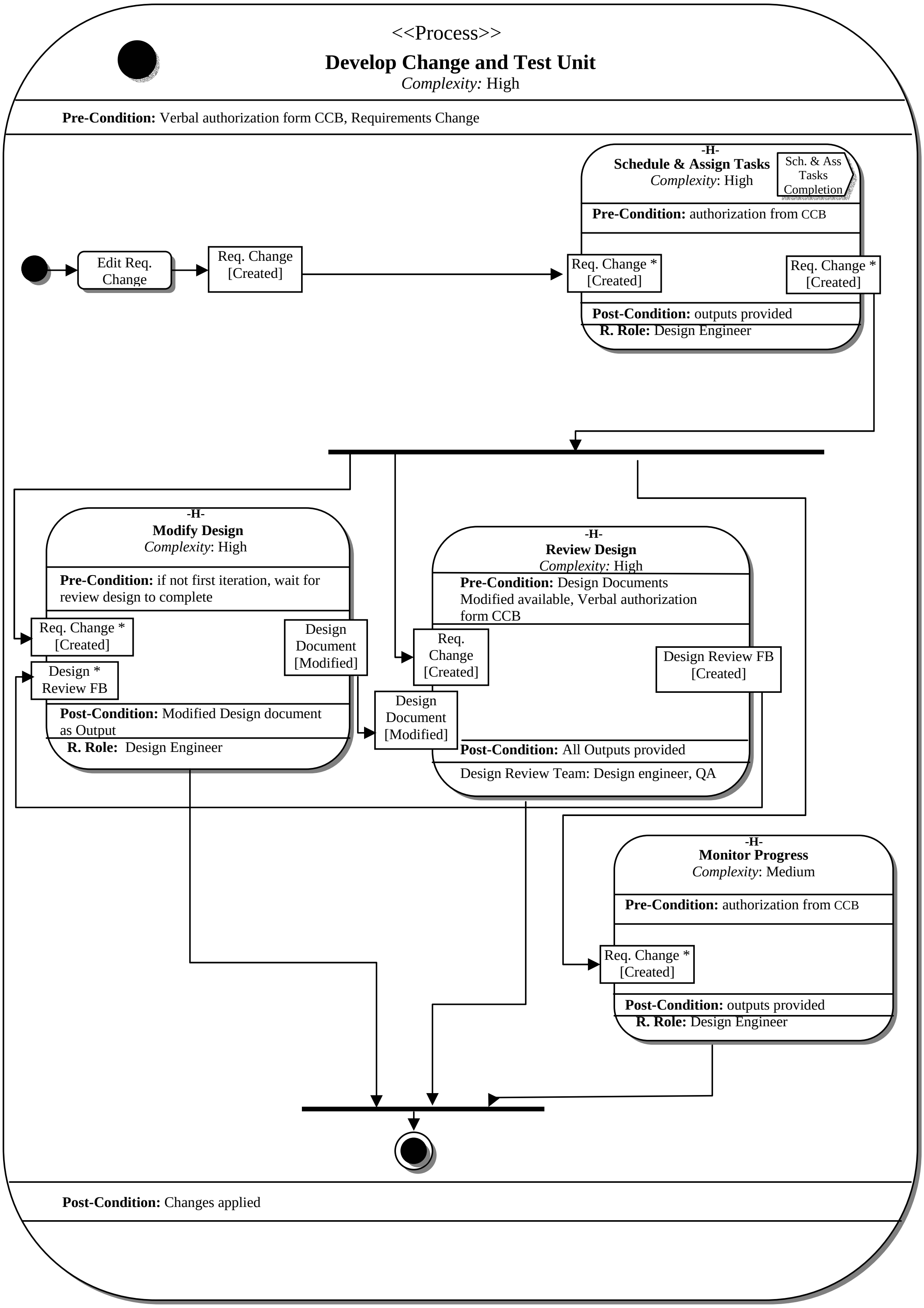
A *Fork* and *Join Nodes* are used to parallelize the execution of different actions. The first flow concentrates on the editing of the Review Report Outcome that will be sent to the *Monitor Progress* task (not represented here). The second flow is conditioned by the result of the review action. For evaluating the result of the review action, a *Decision Node* is used. If the design is approved then, the "Design Document" state is set to "Approved". Otherwise, a document containing the review design feed backs is created and sent to *Modify Design* using an asynchronous *CallBehaviorAction*.

Finally, a *Join Node* is used to synchronize between the different flows before ending the task.

Process View

As discussed in the previous section (c.f. 3.3. Precision), UML4SPM offers an *Activity View* (e.g. figure 6.5. and figure 6.6.) and a *Process View* (figure 6.7). That latter gives an abstract description of the whole process in terms of activities while masking action details. Sequencing of activities, their parameters in terms of *WorkProducts* with their actual states are represented. However, what has to be done within the activity are not represented.

In figure 6.7., a description of a part of the ISPW-6 example is given, but not all activities are represented, just those presented in this section and those that relate to them are depicted. The Develop Change and Test Units activity represents the high level abstraction of the process described in the core problem. It contains all subsequent activities, actions to activate them and initial and final nodes.



<<Process>>
Develop Change and Test Unit
Complexity: High
Pre-Condition: Verbal authorization form CCB, Requirements Change
Edit Req. Change
Req. Change [Created]
-H-
Schedule & Assign Tasks
Complexity: High
Sch. & Ass Tasks Completion
Pre-Condition: authorization from CCB
Req. Change * [Created]
Req. Change * [Created]
Post-Condition: outputs provided
R. Role: Design Engineer
-H-
Modify Design
Complexity: High
Pre-Condition: if not first iteration, wait for review design to complete
Req. Change * [Created]
Design * Review FB
Design Document [Modified]
Post-Condition: Modified Design document as Output
R. Role: Design Engineer
-H-
Review Design
Complexity: High
Pre-Condition: Design Documents Modified available, Verbal authorization form CCB
Req. Change [Created]
Design Document [Modified]
Design Review FB [Created]
Post-Condition: All Outputs provided
Design Review Team: Design engineer, QA
-H-
Monitor Progress
Complexity: Medium
Pre-Condition: authorization from CCB
Req. Change * [Created]
Post-Condition: outputs provided
R. Role: Design Engineer
Post-Condition: Changes applied

Figure 6.7. UML4SPM Process View

3.3. Discussion

In the previous section we saw some activity representations modeled using UML4SPM. Our first goal behind the UML4SPM notation is to make sure that the process modeler or process agent can easily understand what he/she has to do during the development process. This implies a fine-grained description of process's activities. Additionally, if the ultimate goal of the process description is to be executed, the process model has to be precise enough to be mapped to some programming languages or to be directly executed.

While comparing some PMLs that attempted to describe the ISPW-6 example [Kellner 91a], we found that for most of them, process descriptions were restricted to the description of activity resources (i.e., inputs, outputs, agents) and to the way they are coordinated. However, what has to be actually performed within the activity, is not modeled anywhere. Unlike these PMLs, where a software activity is modeled as a black box with just its name and resources attached on it, UML4SPM offers two levels of abstractions. The *Activity View* which allows giving all details about the execution of the activity (parameters, loops, guards, workproducts states, events, etc.) and the *Process View* for a more abstract representation of the process.

The description of the benchmark process by UML4SPM was not just limited to the eight activities of the core problem (not all presented here) but it also succeeded to express most optional extensions.

Regarding the *Core* problem, we succeeded in modeling all the activities using concepts we introduced in Chapter 5 and in Section 3 of this chapter. Regarding optional extensions, which mainly deal with process execution, they were not introduced here. In this chapter we only saw that UML4SPM provides the set of actions and concepts that will allow the specification of executable process models by using for instance, *CallBehaviorActions*, actions related to the raising of exceptions, sending of events, etc. The ways they will be effectively executed are presented in the following chapters

Finally, we also see that thanks to the *DataStoreNode* concept and the possibility to specify WorkProduct states while transferring them along activity nodes, ISPW-6 optional issues related storage and management of WorkProduct can be handled. Optional extensions that deal with dynamic process changes and evolutions are not ensured by UML4SPM.

4. Conclusion

In this chapter, we evaluated UML4SPM through the set of SPML requirements we introduced in Chapter 4. We saw that UML4SPM succeeded in fulfilling the majority of them. Semantic richness is provided thanks to a rich set of process elements we defined in the metamodel, to powerful mechanisms for activity and action coordination and sequencing borrowed from UML2.0, etc. Modularization is addressed by using the *CallBehaviorAction* as means to compose, to call or to coordinate between activity executions. The Precision requirement is reached thanks to the set of concepts such as *Software Activity*, *Action* and *Software Activity Kind* elements which allow the modeling of any process hierarchy. Regarding Understandability, undeniably, UML4SPM has a serious advantage since it reuses UML2.0 notation and diagrams. UML2.0 is wide-spread and many people are already familiar with its use. Finally, we demonstrated the potential and the possibility of UML4SPM for defining executable process models. This is ensured thanks to the set of UML2.0 actions with executable semantics we identified. Execution possibilities and issues are introduced in the following chapters.

Another evaluation of UML4SPM consisted in representing the well-known ISPW-6 software process example using our language. The process example comes in form of core problem and optional extensions. With UML4SPM, we succeeded in modeling all process's activity aspects and issues related to the core problem. Additionally, we addressed main parts of optional extensions that relate to WorkProduct storage and management. Those that relate to process executions are treated in the following chapters (i.e. 7 and 8) while presenting the different approaches we explored for UML4SPM process model executions. Finally, optional extensions related to process changes and evolutions are not ensured by UML4SPM. The evaluation of UML4SPM was validated in [Bendraou 06].

In the following chapters, we address the last part of our proposition which deals with the execution of UML4SPM software process models.

Chapter 7

Execution of UML4SPM Software Process Models: the UML4SPM-2-WSBPEL Approach

1. Introduction

The previous parts of this document focused on giving the state of the art of process technology domains as well as of some significant UML-Based SPMLs. We then introduced our proposal, namely, UML4SPM, a UML2.0-Based language for modeling software processes. UML4SPM promotes understandability, expressiveness, precision and modularization through its reuse of some powerful UML2.0 *Activity* and *Action* concepts. We presented the metamodel of the language, its notation and we evaluated it against main SPML requirements. The next step of our proposal is to deal with the execution of UML4SPM process models. This last part of the document is dedicated to this topic.

For not starting from scratch, we decided to explore the existing propositions in the Business Process Management community. The idea behind is to leverage the maturity level of this domain but most of all, to take advantage of process engines and tooling supports already provided in the BPM field.

To do so, in the following, we will start by giving the main motivations that led us to investigate the possibility of using a business process execution language called WS-BPEL as a target language for executing UML4SPM process models. Our choice of using WS-BPEL is motivated by the fact that recently, WS-BPEL became the de facto standard for process executions. WS-BPEL is introduced in Section 3. In order to transform UML4SPM process models into a WS-BPEL code, we identified a set of mapping rules that we present in Section 4. Human interactions being the heart and soul of software development processes, in Section 5 we address some issues and how WS-BPEL deals with this point. The transformation of UML4SPM to WS-BPEL is given in Section 6. To illustrate the approach, a software process example is used. We also present main steps of the transformation and we discuss some issues related to the transformation. Before concluding this chapter, a discussion and some feed backs about the approach are given in Section 7.

The work exposed in this chapter was realized in collaboration with our industrial partner Softeam, in the context of the MODELPLEX project [Modelplex].

2. Combining UML4SPM and WS-BPEL for Software process model executions

The principal ingredients that participate in the success of UML - among others - are its ability of abstracting the complexity of systems under specification and the fact that the standard provides an intuitive and understandable set of notations and diagrams. This made us exploring the possibility of using UML as a SPML

[Bendraou 05a] as many other approaches (UML1.3 and UML1.4 based) did it before [Jäger 98] [OMG 02] [Di Nitto 02] [Chou 02]. However, whether UML provides a high-level of abstraction and understandability in representing process models, it lacks of some semantics, concepts and tools for their execution [Rumpe 02].

The recently adopted UML2.0 standard brings many new concepts and facilities that make UML suitable for process modeling [Hausmann 05]. These concepts relate to *Activity Diagrams* which provide expressiveness in modeling most complex patterns of control and data flows [Russel 06]. Additionally, the standard provides a set of actions with an executable semantics that allow the sequencing and synchronization between activities, raising exceptions, sending of events, etc. These actions were introduced in Chapter 5 and some examples of their usability were presented in Chapter 6.

Nevertheless, whether the executable semantics of *Activity* and *Action* concepts is provided by the standard in natural language, UML2.0 does not propose a concrete syntax or a proper implementation of these semantics. This has led to the emergence of many propositions of Action Languages. Most popular of them are the *Action Specification Language* promoted by the Kennedy Carter Group [Raistrick 04] and the *Executable UML* initiative promoted by S.J. Mellor et al [Mellor 02].

On the other hand, in the Business Process Management (BPM) domain, recently, a consolidation has led to a single language for business process executions: the Business Process Execution Language for Web Services (WS-BPEL or BPEL for short) [WSBPEL 07]. Rapidly, BPEL gained importance in the industry and became the *de facto* standard for business process orchestrations. Many tool vendors already provide training supports and process engines for this standard [ActiveBPEL] [ApacheAgila]. However, whether BPEL proved to be an efficient executable process language, it remains too low-level to be used for process comprehension and communication between actors of the process. The main design goal of BPEL was to provide a machine-readable language (XML-based) for orchestrating and composing automatically different business processes rather than a language for documenting software development processes.

Instead of reinventing the wheel and in order to execute UML4SPM process models, we decided to explore the possibility of combining both languages i.e., UML4SPM and BPEL. While UML4SPM comes with a high degree of abstraction, expressiveness, concepts and notations suitable for modeling software processes, BPEL provides constructs and precision required for their execution support. Thus, UML4SPM is used as a high-level language for modeling software processes. UML4SPM process descriptions will be then mapped to BPEL in order to be executed.

Our main motivations for combining both languages are:

- To keep a clear separation between the business concerns of software process descriptions (i.e., Phases, Activities, Roles, etc.) and all the technical and organizational features needed for their execution support (Task sequencing, Artifacts assignment, alarms, events and exception handling, etc);

- To leverage the maturity level of the BPM field and the bunch of existing tools instead of starting from scratch. This approach will reinforce the connection between process modeling tools and process execution tools.

In the next section, we briefly introduce WS-BPEL.

3. WS-BPEL2.0

WS-BPEL (Business Process Execution Language for Web Services) is an XML-based standard for specifying the way a set of web services can be orchestrated in order to implement business processes [WSBPEL 07]. It is standardized by the Organization for the Advancement of Structured Information Standards (OASIS) [OASIS].

3.1. Origins

The origins of WS-BPEL result from a collaborative work done between IBM and Microsoft. They both had defined their own orchestrating language, namely WSFL and XLANG (respectively). They then decided to combine them into a new language, BPEL4WS. In April 2003, BEA Systems, IBM, Microsoft, SAP and Siebel Systems submitted BPEL4WS 1.1 [BPEL4WS 03] to OASIS for standardization via the Web Services BPEL Technical Committee. Although BPEL4WS appeared as both a 1.0 and 1.1 version, the OASIS WS-BPEL technical committee voted on 14 September 2004 to name their spec WS-BPEL 2.0. This change in name was done to align BPEL with other Web Service standard naming conventions which start with WS- and accounts for the significant enhancements between BPEL4WS 1.1 and WS-BPEL 2.0. For short, people use to employ BPEL instead of WS-BPEL or BPEL4WS. We will refer to WS-BEPL2.0 as BPEL in the following.

BPEL is built upon WSDL (Web Services Definition Language) [WSDL 01] for describing outgoing/incoming messages between web services. It also uses XML Schema as means to specify variable types [XML Schema 04a, XML Schema 04b].

3.2. WS-BPEL Process

A “program” in WS-BPEL is called a *Process*. A *Process* consists of a set of nested *Activities*. *Activities* fall into two categories: *Basic Activities* and *Structured Activities*.

Basic Activities correspond to atomic actions. This includes *invoke*, for invoking an operation on a web service; *receive*, for waiting a message from a partner; *reply*, for replying to a partner; *assign*, in order to assign a value to a variable; *exit*, for terminating the entire process instance; *empty*, doing nothing; and so on. In WS-BPEL2.0, new activities were introduced such as *if-then-else*, *repeatUntil*, *validate*, *forEach* (parallel and sequential), *rethrow* and *extensionActivity*.

Structured Activities impose behavioral and execution constraints on a set of activities contained within them. These include: *sequence*, for defining an execution order; *flow*, for parallel routing; *switch*, for conditional routing; *pick*, for capturing a race between timing and message receipt events; *while*, for structured looping; and *scope*, for grouping activities into blocks to which event, fault and compensation handlers may be attached [Ouyang 06].

WS-BPEL processes are closely coupled with WSDL. A WS-BPEL process provides a web service interfaces described in WSDL and at the same time deals with services that also have to be described in WSDL. From this point of view, a WS-BPEL process represents a compound web service.

A WS-BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. The description of the deployment of a WS-BPEL process, as well as of WSDL is out of scope of this document.

3.3. WS-BPEL Interaction Model

WS-BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in what is called a *partnerLink*.

The WS-BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. A *PartnerLink* has a *PartnerLinkType*, which defines which WSDL *PortType* is used in a relationship with some partner and which *PortType* is used when a partner interacts with the process itself. These two relationships are defined in the *partnerRole* and *myRole* attributes of the *PartnerLinkType*.

For two-way relationships, both roles are specified. An important aspect is that the use of *PortTypes* means that WS-BPEL only refers to services in an abstract way and it is up to an execution engine to determine which port - and therefore binding - should be used for each *PortType*. Generally, the bindings can be specified statically at deployment time or dynamically – either from within the process or using some engine-specific mechanism [Dobson 06].

WS-BPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Moreover, WS-BPEL introduces a mechanism to define how individual or composite activities within a unit of work are to be compensated in cases where exceptions occur or a partner requests reversal.

3.4. Related XML Specifications

WS-BPEL utilizes several XML specifications: WSDL 1.1 [WSDL 01], XML Schema 1.0 [XML Schema 04a, XML Schema 04b], XPath 1.0 [Xpath 99] and XSLT 1.0 [XSLT 99]. WSDL messages and XML Schema type definitions provide the data model used by WS-BPEL processes. XPath and XSLT provide support for data manipulation. All external resources and partners are represented as WSDL services. WS-BPEL provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

Obviously, the aim of this section is not to present in detail all features of the language. More information as well as examples can be found in [WSBPEL 07].

In the next section, we present mapping rules between UML4SPM concepts and WS-BEPL constructs.

4. From UML4SPM to WS-BPEL

In this section, we address the mapping between UML4SPM and WS-BPEL. Since UML4SPM is UML-based, we start by introducing some related works done in literature concerning the mappings between UML and BPEL. Then, we will present the mapping rules we identified between UML4SPM and WS-BPEL Finally, we discuss some obstacles we faced while establishing these mapping rules.

4.1. UML to WS-BPEL Related Work

In the literature, we can find some works done concerning the mapping of UML Activity diagrams to BPEL. In [Mantell 05], the author defines a UML Profile for automated business processes and maps UML1.4 elements to BPEL1.1. In UML1.4, Activity diagrams were completely different from UML2.0 ones. They were a special case of state diagrams and no actions with executable semantics were provided. This resulted to a very coarse-grained mapping with only few correspondence rules proposed (see table 7.1.).

Profile Construct	**BPEL4WS Concept**
<<process>> class	BPEL process definition
Activity graph on a <<process>> class	BPEL activity hierarchy
<<process>> class attributes	BPEL variables
Hierarchical structure and control flow	BPEL sequence and flow activities
<<receive>>, <<reply>>, <<invoke>> activities	BPEL activities

Table 7.1. UML1.4 Profile to BPEL4WS by [Mantell 05]

With the adoption of UML2.0, Activity diagrams are enriched with executable action semantics. These actions reduced the gap between both languages (i.e., UML2.0 and BPEL.) In [Korherr 06], authors define a UML2.0 Profile for BPEL1.1 and propose a mapping between the two formalisms. However, this was only restricted to actions and did not cover activity elements such as *Fork node*, *Decision node*, *Control Flow*, etc. Similarly, in [Bodbar 04], author concentrated on UML2.0 actions. Mappings for *Control Nodes* (fork, join, merge, etc.), *Loops*, and *Exception constructs* were not defined. Moreover, authors map the UML2.0 *Control Flow* as a BPEL1.1 *Sequence* activity. However, a UML *Control Flow* can only link two activities (i.e., When activity A finishes, B starts). While the BPEL *Sequence* activity defines a block where one or more activities are to be performed sequentially.

4.2. Mapping Rules

Table 7.2 lists major mapping rules we identified between UML4SPM and WS-BPEL2.0. UML4SPM proposes new concepts that deal with the modeling of software process concerns (.i.e., *Roles*, *Guidance*, *Artifact*, *TimeLimit*, etc.) and reuses UML2.0 *Activity* and *Action* package elements, which deal with actions sequencing and synchronization, exceptions, events, invocation, etc.

In the following, we present the set of mapping rules we identified between UML4SPM and WS-BPEL. For their establishment, we've been studying carefully the WS-BPEL specification as well as the UML2.0 standard since we use it as a basis of UML4SPM. Discussion on these mapping rules is given in Section 4.2.

UML4SPM	WS-BPEL2.0
Software Activity	BPEL Process
SoftwareActivityKind	BPEL Variable
Software Activity's attributes and associations	BPEL Variable with name = "attributeName" (respectively "associationEndName") and type. The type may be simple or complex and can be defined in a XML Schema file
Software Activity hierarchy and enclosing elements (actions, inputpins and ouputpin, control nodes, etc)	BPEL Sequence or Flow elements
Pre and Post Conditions of a SA (derived through transitive associations between Activity and Constraint from the UML2.0 metamodel)	BPEL Transition Condition element
Value of the Pre/Post Condition	The text element of the BPEL Transition Condition
WorkProduct input or output of Actions	BPEL Variable with attribute MessageType equals to the WorkProduct Type defined in the WSDL (respectively in the XML Schema). If the Action has more than one WorkProduct than one WSDL Message Part (name=workProductName) with its type is to be defined for each WorkProduct within the MessageType. The attributes of the WorkProduct have to be defined in the type of the WorkProduct in an XML Schema file
WorkProductKind	BPEL Variable
Responsible Role	BPEL Variable
ResponsibleRoleKind	BPEL Variable
TimeLimit of a SA (Associations startsAt, endsAt between a Software Activity and a TimeLimit)	BPEL Variable
Guidance	BPEL Variable
Team	BPEL Variable
Agent	BPEL Variable
Tool	BPEL Variable
AcceptEventAction	BPEL Receive Activity
AcceptEventAction that waits for an event among a list of possible events	BPEL Pick activity. Accepts a message among a list of possible expected messages
AcceptCall Action et ReplyAction to model synchronious calls	BPEL Receive activity with a Reply and input and output specification
Variable (in the context of a StructuredActivity)	BPEL Variable
ReadVariableAction followed by a WriteVariableAction (with an explicit control flow between the Read action and the Write action)	BPEL Assign with From (for reading) and To (for writing) within the Copy element

CallBehaviorAction (Sync / Async)	BPEL Invoke activity (with input and output specification / Only input specification)
CallOperationAction (Sync / Async)	BPEL Invoke activity (with input and output specification / Only input specification)
RaiseExceptionAction. The exception type is defined by the action's InputPin	BPEL Throw activity. Throw has a FaultVariable attribute that corresponds to the exception type
An AcceptEventAction that wait for a TimeEvent	BPEL Wait activity. Waits for a deadline (use of Until element) or a duration (use of For element)
AcceptEventAction	onEvent in the EventHandlers section
OpaqueAction	N/A
InitialNode	BPEL Receive with a "CreateInstance=true"
FinalNode	BPEL Exit activity may be used to abort the process
ControlFlow	BPEL Link element combined with Source and Target elements
ObjectFlow	BPEL Assign with From (the source) and To (the target) within the Copy element
DecisionNode (with control flows that follow the decision node). The Condition is expressed via the association decisionInput:Behavior.	BPEL IF activity witht Condition element to express the condition
ExceptionHandler	BPEL FaultHandlers with Catch
ForkNode to express parallelism.	BPEL Flow Activity
JoinNode	BPEL Link element combined with Source and Target elements
LoopNode with Test expressed via the association test:ExecutableNode	While activity with element Condition
	RepeatUnitl activity
	ForEach activity
StructuredActivity (defines an activity with its actions, control nodes, variables limited to the activity scope, etc.)	BPEL Scope Activity with all its partnerlinks, variables, faulthandlers, etc.

Table 7.2. UML4SPM to WS-BPEL2.0

4.3. Discussion

While establishing these mapping rules we have noticed many observations. The most important one relates to the fact that all elements in UML4SPM that provide semantics proper to software process modeling have no equivalent in WS-BPEL. All elements such as *Responsible Role*, *Guidance*, *Time Limit*, etc are converted to BPEL process variables. On the other hand, all elements dealing with the coordination of activities, events, exception handling, etc. map easily to WS-BPEL concepts. This observation comforted us in our choice of combining the two languages, UML4SPM as a language providing high level abstractions for process modeling and communication, and the WS-BPEL for process execution.

The second observation is that there is no one-to-one correspondence between UML4SPM elements and WS-BPEL elements. As we can see in table 7.2, an UML4SPM element (e.g., *LoopNode*) can be mapped into different WS-BPEL elements (i.e., *While*, *Repeat Until*, or *ForEach* activities). This implies that during the transformation phase, the process modeler has to choose one mapping rule among those proposed (if multiple choices) and always apply the same one along the process specification. On the other hand, there are some WS-BPEL concepts that have no equivalent in UML4SPM such as *Validate*, *Empty*, or *Extension Activities*.

Another important issue relates to the impossibility of WS-BPEL to support some Control Flow patterns, more commonly known as workflow patterns [Van der Aalst 03a]. Indeed, BPEL lacks support of multiple merges pattern (merge many execution paths without synchronizing) and discriminators pattern (merge many execution paths without synchronizing. Execute the subsequent activity only once). It also does not allow the synchronization of multiple instances of the same activity and lacks support of arbitrary cycles. These lacks then have to be taken into account while modeling software processes with UML4SPM in order to avoid the use of patterns that are not supported by WS-BPEL. To avoid for instance arbitrary cycles we propose to combine the use of a *SendSignalAction* and an *AcceptEventAction*. These concepts can be an alternative to cycles and map to WS-BPEL concepts (*Invoke* and *Receive* activities for instance).

Finally, the last issue relates to human interactions within the process execution and which we address separately in the next section.

5. Human Interactions

While some business processes can be fully automated, software processes are composed of creative activities (e.g., modeling, checking, communicating, decisions, etc.) that make them need a support for human interactions. Even in the field of BPM, it has been recognized that the human dimension is essential for process realization. We can notice in Table 7.2 that WS-BPEL does not provide any support for this kind of activities.

The interaction scenarios can be very simple, like manual approval. However, more complicated scenarios like entering data, assignment of tasks to other users and managing long-running processes may appear. In UML4SPM, we have the possibility to express that an activity is automated or has to be carried out by a human. Opaque Actions are also used to model manual tasks and human interactions. One solution would be to map this data as WS-BPEL process variable that the process engine can take into account at enactment time. However, this would not be a long term and reusable solution, especially if we have to deal with complicated interactions.

In the BPM domain, the current state of the art clearly distinguishes two approaches that aim in solving the human interaction issue. In the following, we present them.

5.1. The Workflow Service

To face WS-BPEL's lack in supporting human interactions within processes, many vendors have solved this problem by implementing manual tasks as a "normal" asynchronous Service (from the perspective of the orchestrating process). The WS-BPEL keeps its original functionality and does not need any extensions, but each task needs its own interface specified in the WSDL. Whenever a task needs knowledge about the process state it must be passed within the invoke call to this task. This means a lot of manual work must be done for each task.

To solve this problem the notion of *Workflow Service* is introduced [Juric 07]. This service can be called asynchronously from WS-BPEL to perform operations like adding, updating, completing, renewing and routing tasks. User applications, on the other hand, can communicate with the *Workflow Service* to acquire the list of tasks for selected users, render appropriate user interfaces, and return results to the *Workflow Service*, which forwards them to the BPEL process. A schematic overview is given by [Juric 07] in Figure 7.1.

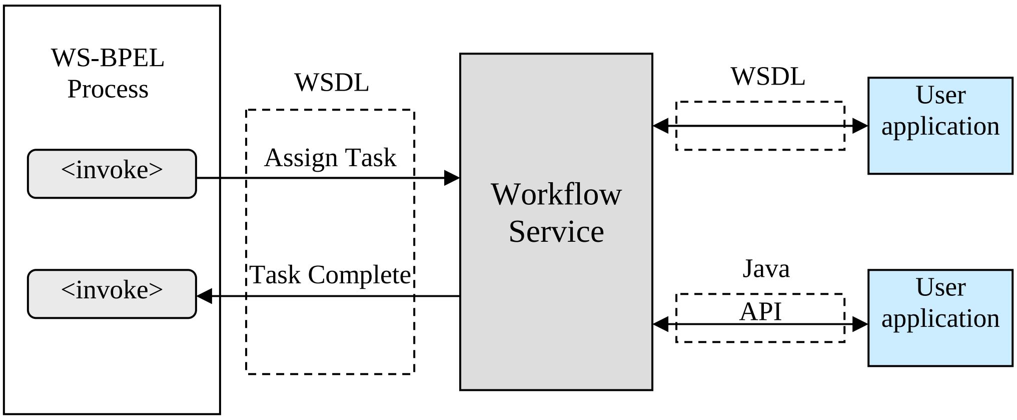


Figure 7.1. Workflow integration with WS-BPEL

Obviously, the advantage of this approach is that the WS-BPEL standard is not modified. The *Workflow Service* can be implemented within an application server as long as it provides a WSDL interface to the WS-BPEL process engine. The WS-BPEL process doesn't have to know how user tasks are handled. This gives the opportunity to use various kinds of communication channels with users. However, no standard exists specifying the interface of such a *Workflow Service*. Each vendor implements its own, resulting in non-portable WS-BPEL processes which may penalize this approach.

5.2. BPEL4People

In order to deal with the human interaction issue, we decided also to explore a very interesting proposition introduced by industrials (i.e. IBM and SAP) known as "BPEL4People" [Kloppmann 05].

In BPEL4People, a new BPEL activity called *People* activity is introduced. A *People* activity is a basic activity, which is not realized by a piece of software but an action performed by a human being. It can be associated with a group of people, a generic role, etc. BPEL4People describes the following generic human roles interacting with processes:

- *Process Initiator*, the person who actually creates an instance of the process
- *Process Stakeholder*, a person who can influence the progress of a process instance
- *Potential Owner*, a person who can claim and complete a people activity
- *Business Administrator*, is defined for a process and can perform administrative actions on the business process, such as resolving missed deadlines

These generic human roles are associated with a group of people by a so-called *people link*. A *people link* commonly contains a query against an organizational directory in order to determine the actual individuals with which it is associated. The actor of a *People* activity is determined by a *people link*. A *People* activity can be associated with different groups of people, one for each generic human role.

People activities have the same properties as standard BPEL activities, but their implementation is different. *People* activities are implemented by tasks. So instead of invoking some kind of web service, a BPEL engine (actually, an extended BPEL engine implementing BPEL4People) must create a *task* for a certain user. These *tasks* are indivisible units of work performed by a human being. They specify an action that a user must perform. Some properties of a *task* are a description, a priority, expected data, a deadline and a user interface [Kloppmann 05].

The extended BPEL engine creates for each *People* activity - depending on its contents - a list of tasks, also called work items ("to-dos") and affect them to the appropriate process participants. A generic user interface is associated with each task of the activity in order to highlight inputs/outputs of the activity, deadlines, to add the possibility to attach other materials (e.g., guidelines) and to ease communication between agents.

Regarding the implementation of tasks, BPEL4People leaves the choice to the modeler between five possible configurations (see figure 7.2). These five configurations fall into two kinds: *Inline Tasks* and *Standalone Tasks*.

Inline tasks are defined as part of the *People* activity or of the BPEL process (they have access to the process context, variables, etc.) while *Standalone tasks* are defined outside the process.

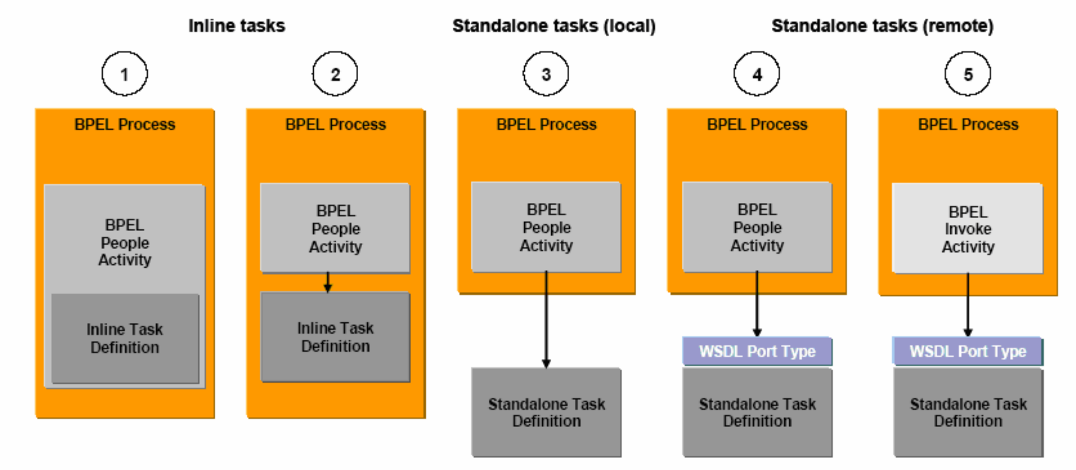


Figure 7.2. BPEL4People Models for Implementing Human Interactions from [Kloppmann 05]

An inline task can be defined in a people activity (model 1 in figure 7.2) or as a top-level construct of the BPEL process (model 2). In this case, the same task can be used within multiple *People* activities. Both models have the advantage of the possibility of context sharing between task and process. This can be used, for example, to implement the "Chained Execution" interaction pattern which consists in the ability of the WS-

BPEL engine to automatically start the next work item in a case once the previous one has completed. The process knows who has performed the previous tasks and can then assign the next task to the same person.

Standalone tasks may be accessed through 1) implementation-specific invocation mechanisms (i.e., no WDSL), 2) a Web service interface defined with WSDL or 3) a BPEL *Invoke* activity that calls a Web service implemented by the task (WSDL + binding).

Model 3 shows an implementation specific definition of a task outside a process without a specific interface. Thus leaving all the communication between process and task implementation specific. In that way application vendors can expose their functionality as tasks which can be called from process engines.

In Model 4 the task is also defined standalone, but with an interface specified using WSDL. This is a more generic approach but an extra standard is needed to propagate state changes between process and task. This "coordination protocol" can be used for example, for performing life cycle operations on the task, such as terminating it. However, BPEL4People authors didn't specify a protocol solving this problem.

Model 5 is the most generic case. In this case the task is called with a standard WS-BPEL *invoke* activity. The BPEL standard remains unharmed, but each task needs its own WSDL definition. The advantage of this model is that, unless the WSDL definition stays the same, the implementation of the task can change. It is even possible to replace a human task with a business rule without the need to change the calling WS-BPEL.

For tool vendor who wants to implement BPEL4People, all these models have to be supported by the process engine.

5.3. Discussion

Looking at the human interaction problem, we can obviously conclude that there is a real need for standardizing this issue in WS-BPEL. Adopting the first approach i.e., *Workflow Service* means no standardization at all, since each vendor defines its own *Workflow Service* depending on its specific needs.

BPEL4People on the other hand comes with a more long term solution and provides a solid base. However, the big issue with BPEL4People is that it extends the existing WS-BPEL standard with new activities and task definitions, extensions that not all tool vendors are ready (agree) to adopt. Currently, existing WS-BPEL tools that handle human interactions do it without using BPEL4People. Instead, they provide a specific web service that manage human tasks and which may be invoked from a WS-BPEL process. Of course, this remains proprietary solutions. When writing this document, some tool vendors claimed their intention to incorporate the BPELPeople proposition in their tool suites. We can cite the Intalio's BPMS Community Edition [Intalio], IBM's WebSphere Process Server [Websphere] and Oracle's BPEL Process Manager [Oracle].

Another obstacle in using BPEL4People is that it imposes the implementation of the five interaction models.

A compromise between both approaches would be more appropriate. This would consist in standardizing the port Type and basic features of the so-called *Workflow Service*. Defining its standard interface should be done in WSDL. Once the *Workflow*

Service is standardized, WS-BPEL processes will be then portable between different vendors. However, vendors can still implement the *Workflow Service* in their own way and integrate it with application servers, and so on, as long as they apply to the standard interface definition.

6. Transforming UML4SPM Process Models to WS-BPEL2.0

As we said in the introduction of this chapter, this work is done in collaboration with our MODELPLEX industrial partner Softeam. Our intention was more to explore the feasibility of the UML4SPM-2-WSBPEL2.0 approach than to provide a fully implemented prototype. We started from a simple software process example described in a natural language. This process example is then modeled using UML4SPM. Finally, a transformation is carried out in order to get the WS-BPEL2.0 code. In the following, we present each of these steps.

6.1. Software Process Example

In this section we introduce a simple yet representative example of a portion of a software development process. This process example was provided by our industrial partners within the IST European Project MODELPLEX, which this work is part of [MODELPLEX 06]. The process example will be first described in natural language and then represented using UML4SPM.

The process is composed of two phases: "Inception" and "Construction" phases. In this document we only address the "Inception" phase. The "Inception" phase is composed of two activities. The "Elaborate Analysis Model" activity and the "Validate Analysis Model" activity.

The "Elaborate Analysis Model" activity takes as input "Requirement Documents" (i.e. work specifications) and produces a UML "Analysis Model". The "Analysis Model" is then taken as input by the "Validate Analysis Model" activity which is composed of the following steps: 1) Check the UML Analysis Model; 2) Edit a Validation Report. If the "Analysis Model" is valid then send an email to the development team and go to the next phase. If the "Analysis Model" is invalid, then send an email to the development team informing them that the validation of the UML Analysis Model failed and terminate the activity. The role in charge of both activities of this phase is ensured by the "Analyst".

Looking at the process description we can notice some aspects that characterize software development processes. The first one is the hierarchy of the process. We have a Phase, which may contain Activities, which in their turn may contain steps. The second aspect is the presence of both human activities and automated activities, which makes it difficult to automate the entire process. Finally, the transformation process of artifacts from one activity into another and the necessity to know the artifact's state at any time of the process.

6.2. The Software Process Example Modeled Using UML4SPM

The process description focuses on the "Inception" phase and activities it owns (figure 7.3a, 7.3.b and 7.3.c).

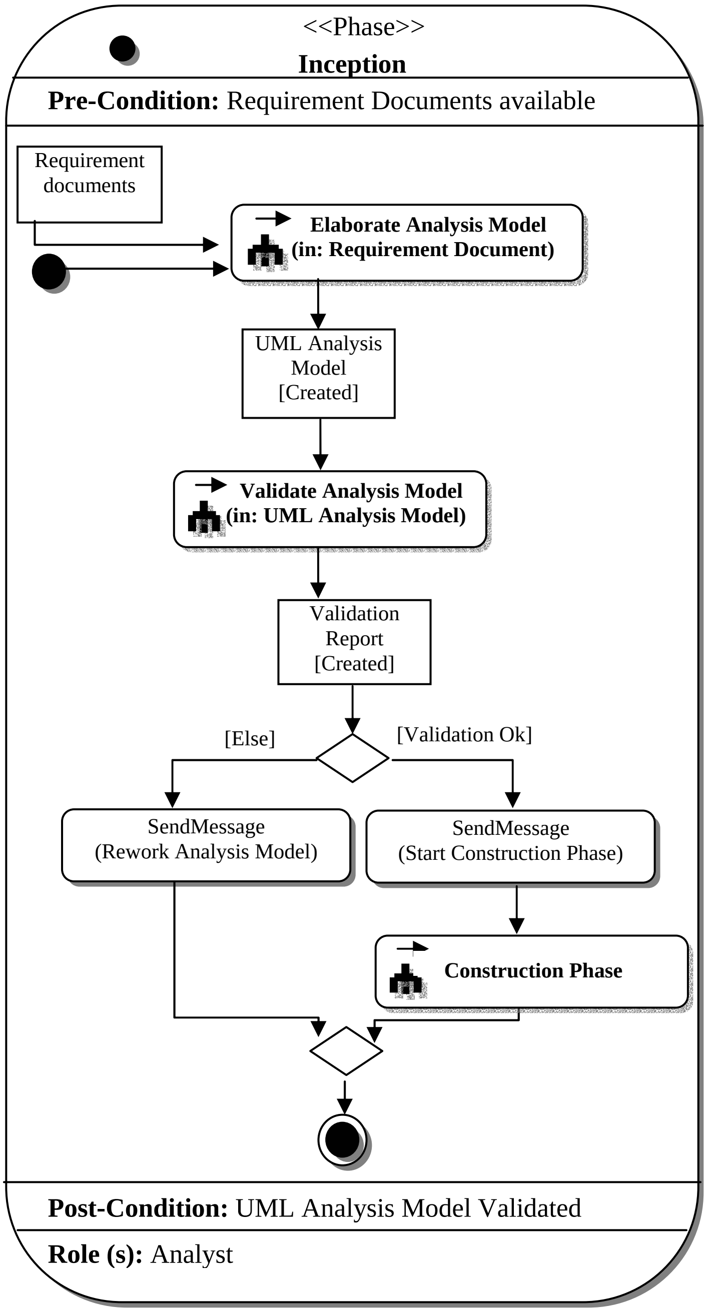

<<Phase>>
Inception
Pre-Condition: Requirement Documents available
Requirement documents
Elaborate Analysis Model (in: Requirement Document)
UML Analysis Model [Created]
Validate Analysis Model (in: UML Analysis Model)
Validation Report [Created]
[Else]
[Validation Ok]
SendMessage (Rework Analysis Model)
SendMessage (Start Construction Phase)
Construction Phase
Post-Condition: UML Analysis Model Validated
Role (s): Analyst

Figure 7.3.a The "Inception" phase

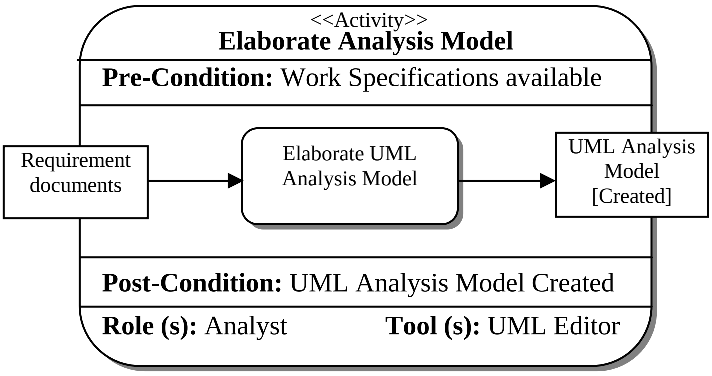
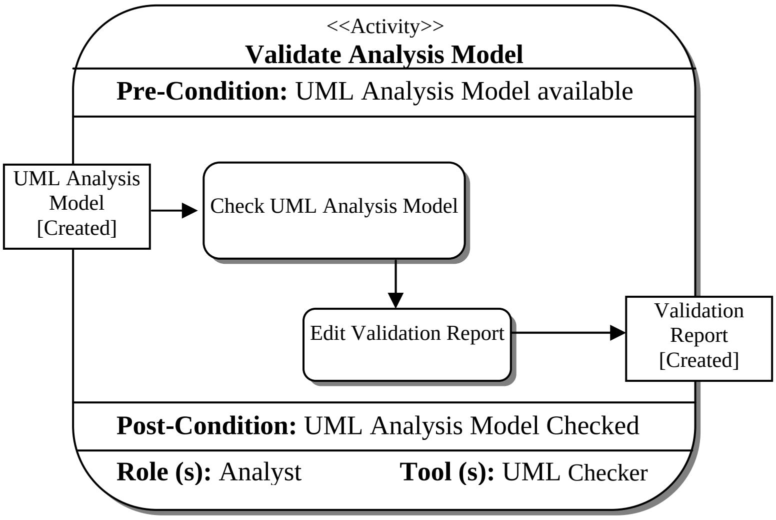

<<Activity>>
Elaborate Analysis Model
Pre-Condition: Work Specifications available
Requirement documents
Elaborate UML Analysis Model
UML Analysis Model [Created]
Post-Condition: UML Analysis Model Created
Role (s): Analyst
Tool (s): UML Editor

Figure 7.3.b The "Elaborate Analyze Model" activity



<<Activity>>
Validate Analysis Model
Pre-Condition: UML Analysis Model available
UML Analysis Model [Created]
Check UML Analysis Model
Edit Validation Report
Validation Report [Created]
Post-Condition: UML Analysis Model Checked
Role (s): Analyst
Tool (s): UML Checker

Figure 7.3.c The "Validate Analysis Model" activity

As we can notice, the "Inception" phase activity represents the context of this process. This is indicated by the start-blob in the top-left corner. It is used to coordinate between different activities and workproducts of the process.

One important aspect is the use of *CallBehaviorActions* in order to initiate/call process's activities (e.g., "Elaborate Analysis Model" call). In the call, we have to precise 1) whether the call is synchronous (use of a compete arrow in the top-left corner) or not (half arrow, e.g., "Construction Phase" call); 2) the parameters of the call, which represent workproducts inputs/outputs of the activity. The parameter types may be *in*, *out* or *inout*.

Another aspect is the use of *Decision*, *Merge* and *Join* nodes. The decision node allows expressing a choice of actions to do depending on a condition (in this case whether the analysis model is valid or not). The merge node here is used to regroup the two branches coming out of the decision node. Whatever the branch, the first one that will reach the merge node will end-up the activity execution. Finally, the join node is used to synchronize between the control flow starting the activity and the availability of the Requirement Document before calling the "Elaborate Analysis Model activity".

In the following, we describe the transformation of UML4SPM models into WS-BPEL2.0 code.

6.3. Transformation

For experimentation purposes, the transformation of UML4SPM process models into WS-BPEL2.0 code is currently carried out by a Java program. However, we plan to formalize the transformation with a model transformation language such as ATL [ATL 06]. The Java program takes as input the UML4SPM model and generates the corresponding WS-BPEL2.0 code. Hereunder, we present in natural language main steps of the transformation algorithm.

In our transformation algorithm we concentrate on the target model by creating WS-BPEL description parts one-by-one and extracting required information from the UML4SPM model. As a typical WS-BPEL description, the target model contains the following parts:

- WSDL imports – for declaring involved web services. By default this part contains “Workflow Administration” service declaration which corresponds to the notion of "Workflow Service" introduced in Section 5.1.
- Variables – data used in the process
- Flow and Sequence– containing activities, service invocations, receive, reply.
- Links – for transition declarations
- Event Handlers – processing incoming events

These parts are filled according to the mapping we defined in table 7.2. In the following, we give the main lines of our algorithm.

- The transformation algorithm starts by the creation of an empty WS-BPEL process definition from a template. The template contains an import for the WSDL description and the partner link definition for the “Workflow Administration”;

- Generation of the "import" and "variable" section. All UML4SPM elements in table 7.2 that map to a WS-BPEL variable are processed here. Variables are created for each software activity, for all its attributes, responsible roles, guidance, used tools, etc. and for storing input and output WorkProducts if any;

- Then, the "flow" or "sequence" section is created (depending of the process model if it starts by a parallel flow or a sequence flow) followed by the "links" declaration. All UML4SPM control flows are generated as WS-BPEL "links" and the *Source* and *Target* elements are documented;

- The WS-BPEL "flow" or "sequence" activity initiating the process starts with a "receive" activity, which is used for communicating input Work Products to the BPEL process. This activity should also contain "createInstance" attribute equals to "true" to indicate that the process is instantiable;

- “Human” actions (defined as opaque action within UML4SPM process models) are transformed into a pair of linked "invoke" / "receive" activities implementing an asynchronous call of "Workflow Administration" web service which is the service dealing with human interactions.

- The remaining UML4SPM elements are transformed according to what was defined in Table 7.2;

- Finally, the "import" section is filled manually in order to document the "partner link" and the WSDL location of web services the process uses, in particular here, the "Workflow Administration" web service.

After applying the algorithm, the transformation results in a WS-BPEL process containing the following WS-BPEL activities:

- "Receive" for instantiating the process and getting Requirement Documents as input;
- "Invoke"/"Receive" for Analysis Model Elaboration – "Invoke" for calling Workflow Administration service providing Analyst with Requirement Documents. "Receive" for getting back an Analysis Model;
- "Invoke"/"Receive" for Analysis Model Validation – "Invoke" for calling Workflow Administration service providing Analyst with the Analysis Model. "Receive" for getting back a Validation Report;
- "if" activity for assessment of the validation result.
- "Invoke" for informing Project Manager and Analyst about results of the software activity.

The generated WS-BPEL process is to be deployed with a conventional BPEL engine, ActiveBPEL in our case [ActiveBPEL]. Then, the process is run according to the WS-BPEL process definition. All human tasks are to be redirected to the "Workflow Administration" web service which provides a console for guiding the agent in performing the task. Listing 1 given in Appendix B of this document gives a sample of the generated process example defined using UML4SPM (presented in section 6.2).

7. Discussing the Approach

Whether WS-BPEL provides a rich set of concepts for executing processes, it lacks of the abstraction and expressiveness needed in modeling human-readable and understandable process definitions. Its deficiency in supporting some workflow patterns, the lack of graphical notation and its no support for human interactions and arbitrary cycles makes it inappropriate for the modeling, communicating and understanding of software processes. On the other hand, UML4SPM provides a high level of abstraction, expressiveness, notation and a set of elements and concepts with executable semantics; however it lacks of enactment support. In this chapter we demonstrated how the two languages are combined in order to complement each other and to fully support both process modeling and execution.

However, even if this approach presents the advantage of leveraging existing BPEL process engines and takes advantage of the execution support, it still suffers from some issues.

The first one deals with the fact that during the transformation process all the aspects and semantics proper to software process activities (roles, guidance, deadlines, etc) are lost or scattered as BPEL variables. The only concepts that have equivalents in BPEL are those that deal with the sequencing of activities, events headlining, etc and which already have executable semantics (i.e.,UML2.0 *Activities* and *Actions*). This has as direct effect, the loss of data needed for process measurement and improvement.

Another issue is that the process modeler has to choose the right concepts, which can be mapped to WS-BPEL while modeling the process. Otherwise, there will be no support for them. As we saw in the mapping rules we defined, some UML4SPM concepts may have many corresponding WS-BPEL elements. Thus process modelers have to choose one of them and to make sure that it will always use the same corresponding WS-BPEL element for a given UML4SPM concepts. Some UML4SPM elements don't have equivalent in WS-BEPL. This also has to be taken into account while modeling software process models using UML4SPM.

Finally, the last issue relates to the fact that the generated WS-BPEL is not usable straightforward after the transformation. A configuration step is needed in order to set *Partner Link* properties (service locations used by the process). This step can be automated during the transformation and process modeler would be asked for instance to enter these information However, if the process modeler adds new elements or variables for execution aims after the transformation, this would raise the issue of traceability between UML4SPM process definition and the generated WS-BPEL code, and how coherence between the two definitions can be preserved.

8. Conclusion

In this chapter we explored the feasibility of using WS-BPEL as a target execution language for UML4SPM process models. At this aim, we introduced the language and its main features. We also gave detailed mapping rules between UML4SPM concepts and WS-BPEL constructs. While identifying these mapping rules we raised some issues. Main ones related to the fact that UML4SPM elements with a semantic proper to software process modeling (i.e., WorkProduct, Role, Guidance, etc) have no equivalent in WS-BPEL. They are only represented as process variables. Also, we raised the fact that there is no one-to-one correspondence between elements of the two languages and BPEL's lack in supporting some control flow patterns. This imposes a certain rigor while modeling the software process using UML4SPM (i.e., some concepts are not represented in BPEL, avoiding arbitrary cycles, in case of multiple mappings always use the same, etc.). The mapping rules we proposed are not only UML4PSM-to-WS-BPEL specific since all rules that deal with UML2.0 concepts can be reused by any UML2.0-Based language or profile for business process modeling.

Another important point we addressed in this chapter was human interactions. We presented the different propositions that can be applied with WS-BPEL in order to take into account the human dimension. We also discussed the advantages and lacks of each approach.

For illustrating the approach, we gave a software process example that we modeled using UML4SPM. We then presented the main steps transforming the UML4SPM process model into WS-BPEL code.

Finally, we discussed the different issues of the approach. We can sum them up in the following points. The first point is the unquestionable advantage of to be able to reuse the myriad of WS-BPEL process engines and training supports provided by the Business Process Management community. The field is very mature and very active, which opens very large perspectives. We don't have to deal with all issues related to resource management, distribution, exceptions, etc. All these aspects are already implemented within process engines.

Another advantage of this approach is to hide the complexity of process executions by using UML4SPM as SPML for communicating, exchanging and understanding of the process. Thus, a clear separation between the business aspects of software processes and their execution is allowed.

On the other hand, the approach presents some lacks. The first one deals with the lack of WS-BPEL in supporting human interactions. Even if we presented some initiatives, no one succeeded to get standardized or fully adopted by tool vendors. Another lack deals with the fact that sometimes process modelers may have to modify the WS-BPEL code for execution purposes which may raise the problem of how these changes will be traced-up to the UML4SPM process models. In case of exhaustive modifications this may lead to incoherent versions between the UML4SPM process model and the WS-BPEL code. This approach also imposes that process modelers have to learn BPEL and to be familiar with its constructs in order to maintain or to modify the generated code.

This approach was validated in [Bendraou 07c] and is currently under evaluation within the MODELPLEX [MODELPLEX 06]. Future perspectives of this work are the formalization of the transformation by means of well-established model transformation languages such as ATL [ATL 06] or QVT [OMG 05b]. This will reduce human intervention and ambiguities due to multiple mappings that one UML4SPM element may have into BPEL. In addition, the support of OCL2.0 as a language for the specification of Pre and Post condition is underway. When writing this chapter, the implementation of the GUI was not yet provided by our partner Softeam which was in charge of realizing the "Workflow Administration" service.

Chapter 8

Execution of UML4SPM Software Process Models: the UML4SPM Executable Model Approach

1. Introduction

In the previous chapter we presented an approach for UML4SPM process model executions. It consisted in transforming UML4SPM process models into WS-BPEL in order to execute them. As we highlighted it, this approach presents the advantage of leveraging the existing business process engines. Aside from that, it suffers from several lacks. Most important ones in our view are:

- BPEL's lack in representing UML4SPM concepts having semantics proper to software process modeling (i.e., Responsible Role, Guidance, Time Limit, WorkProduct, etc.);

- Process modelers have to deal with two languages, UML4SPM and BPEL. Additionally, for any changes in the process model, a new BPEL code generation has to be carried out and a configuration phase is required before deploying the process;

- Any modification in the BPEL code can't be traced-up to the UML4SPM process model which may lead to incoherencies between UML4SPM process models and the generated BPEL code.

All these obstacles can be surmountable if the process of transforming UML4SPM process models into BPEL has to be carried out only once. However, since software processes evolve rapidly, process models have to be updated frequently which requires a new BPEL code generation each time the model is modified. Moreover, sometimes the process currently under execution has to be modified at runtime in order to take into account some new requirements or to react to urgent situations. Definitely, the UML4SPM-2-BPEL approach can't be a long term solution since it can't answer this kind of flexibility.

In this chapter we present a new approach for executing UML4SPM process models. It aims at overcoming the UML4SPM-2-BPEL approach lacks by introducing an *Execution Model* for the UML4SPM Metamodel. The goal of the *Execution Model* is that once process modelers have defined their UML4SPM process models, they just can run them without any intermediate step.

In the following, we start by presenting the UML4SPM execution model approach. Then, in Section 3, we present each of its classes and we go deeply in details through the implementations we propose. The UML4SPM *Execution Model* is then used as basis of the *Process Execution Engine* we propose in section 4. We also introduce the UML4SPM *Process Model Editor* and we give an example of UML4SPM process model execution. Section 5 summarizes all the important aspects of the approach and discusses them. Section 6 concludes this chapter and draws some perspectives.

2. UML4SPM Execution Model Approach

The UML4SPM *Execution Model* tends to bring life to elements of the UML4SPM metamodel. By life, we mean a precise specification of the runtime behavior of each element of the metamodel. Therefore, a UML4SPM process model once edited can be straightforward executed upon a simple click without any additions or intermediate steps. The only condition is that process models are well formed. By well formed, we mean that the model should respect the structure and constraints defined in the metamodel. It also supposes that the process model is complete in the sense that it specifies a coherent sequence of actions, control nodes, object nodes, etc that allows its execution. For instance, a software activity, without an initial node and without activity parameter nodes can never be started. A process model containing several software activities with no one with its "*isInitial*" attribute set to "true" also will never be launched since we need one and only one initial software activity within the process.

The idea of the *Execution Model* is inspired from the RFP (Request For Proposal) issued by the OMG called: *Executable UML Foundation* and which LIP6 is part of the standardization working group [OMG 05c]. The objective of this initiative is the definition of a computationally complete and compact subset of UML 2.0 to be known as “Executable UML Foundation”, along with a full specification of the execution semantics of this subset. “Computationally complete” means that the subset shall be sufficiently expressive to allow definition of models that can be executed on a computer either through interpretation or as equivalent computer programs generated from the models through some kind of automated transformations. The execution semantics of this subset is based on the semantics of UML2.0 elements, which is given in natural language in the standard.

Since that the building blocks of UML4SPM are UML2.0 Activity and Action packages, we found it interesting to take advantage of the execution model proposed by the *Executable UML* specification, while focusing on UML2.0 elements we reused in our SPML. In UML4SPM, activity elements and actions are used for sequencing the process's flow of work and data, for expressing actions, events, decision, concurrency, exceptions and so on. Thus, the implementation of the execution behavior for these concepts will be used as the core engine of UML4SPM. While writing this document, there was only one Executable UML submission document which is rather a draft than a complete specification. Thus, we reused some parts of the executable model proposed, we modified others and we add what we believed essential and which was lacking by the model. We will detail these concepts while presenting realization steps in the following sections.

The *Executable UML* specification introduces the execution model in form of class diagrams; each class represents the executable class of a UML element. By executable class it is meant, a class having a set of operations aiming at describing the execution behavior of the UML element as it is specified in the UML2.0 standard in natural language. However, the specification does not offer an implementation of these operations. The one who wants to use the executable model still needs to implement each of the operations defined within the class diagrams with respect to the semantics defined in the UML2.0 standard. In the context of UML4SPM we did implement these operations in order to realize our process engine. The implementation of the UML executable model was restricted to Activity and Action elements we reused within UML4SPM and respects the UML2.0 semantics.

In the following we introduce the UML4SPM *Executable Model* before going through the realization steps of the UML4SPM process engine.

3. The UML4SPM Executable Model

The *Executable UML Foundation* specification defines the minimal subset of UML metamodel elements that allows the specification of complete and executable models. By complete we mean all the required metaclasses, properties and associations representing the necessary data for a UML model to be executed. This subset is called UML *Foundational subset (fUML)*. The contents of the foundational subset are determined partly by two opposing criteria:

- Compactness: The subset should be small. This facilitates definition of a clear semantics, and implementation of execution engines;

- Ease of translation: The subset should enable straightforward translation from common surface subsets of UML to *fUML* and from *fUML* to common computational platform languages.

For the definition of *fUML,* simplifications are carried out upon the UML2.0 metamodel. These simplifications span, properties, associations, methods, but also some metaclasses. For instance the *CallEvent*, *ChangeEvent* and *TimeEvent* metaclasses are not taken into account considering that a *SignalEvent* can be used for all situations. The same thing is applied to *DecisionNode*, claiming that a *ConditionalNode* can be used instead. In the context of UML4SPM, we cannot manage with these kinds of simplifications since we use these concepts for process model definitions. We will see for instance that in the case of *DecisionNode*, we define an execution class for this element, a set of operations and their implementations describing the *DecisionNode* behavior according to the UML2.0 semantics. More details on simplifications operated on the UML2.0 metamodel in order to obtain the *fUML* can be found in [OMG 06e].

The next subsection presents the rationale of the executable model before introducing its different classes and operations.

3.1. Executable Model: Rationale

The *Executable Model* we defined for UML4SPM specifies the precise execution behavior of UML4SPM metamodel elements. It takes as a basis the one defined in the *Executable UML Foundation* specification and adds some features (properties, operations and classes) and semantics proper to software process modeling.

The execution model is inspired from the GoF Visitor pattern [Gamma 94]. The idea is to decouple the elements defined in the UML4SPM metamodel from their runtime behavior. Thus, for each element in the UML4SPM metamodel for which a behavior is to be defined, there is a runtime “Execution” visitor class in the execution model that represents a single execution of that element. Therefore, we will have for the *Software Activity* element, an *ActivityExecution* class, for the *ActivityNode* an *ActivityNodeExecution* class, for the *ForkNode* a *ForkNodeExecution* class, and so on. Each class having a set of operations that once implemented, reproduce the execution behavior of the element. The Visitor pattern typically requires implementation of a “visit” operation on the visitor class and an “accept” operation on the visited class. In the execution model, execution classes have an association that points to the UML4SPM element to which they add behavior. This is in line with the purpose of the Visitor pattern which “represents an operation to be performed on the element(s) of an object structure” and allows the addition of behavior to the elements in UML4SPM without actually modifying them.

Figure 8.1 draws the big picture of the execution model principle by giving the example of *Software Activity*, *Activity Edge* and *Activity Node* elements and their corresponding executable classes in the execution model. The operations defined in the executable classes slightly differ from those specified in the Executable UML specification. We added new operations in order to take into account some software process modeling aspects or because that during the realization phase, we realized that they were lacking. We ignored others that we found useless in the context of UML4SPM, or we redefined them for execution purposes.

While presenting the different execution classes, we will highlight the important features of this approach and of the implementation we provide. We will also give details on operations and features we added.

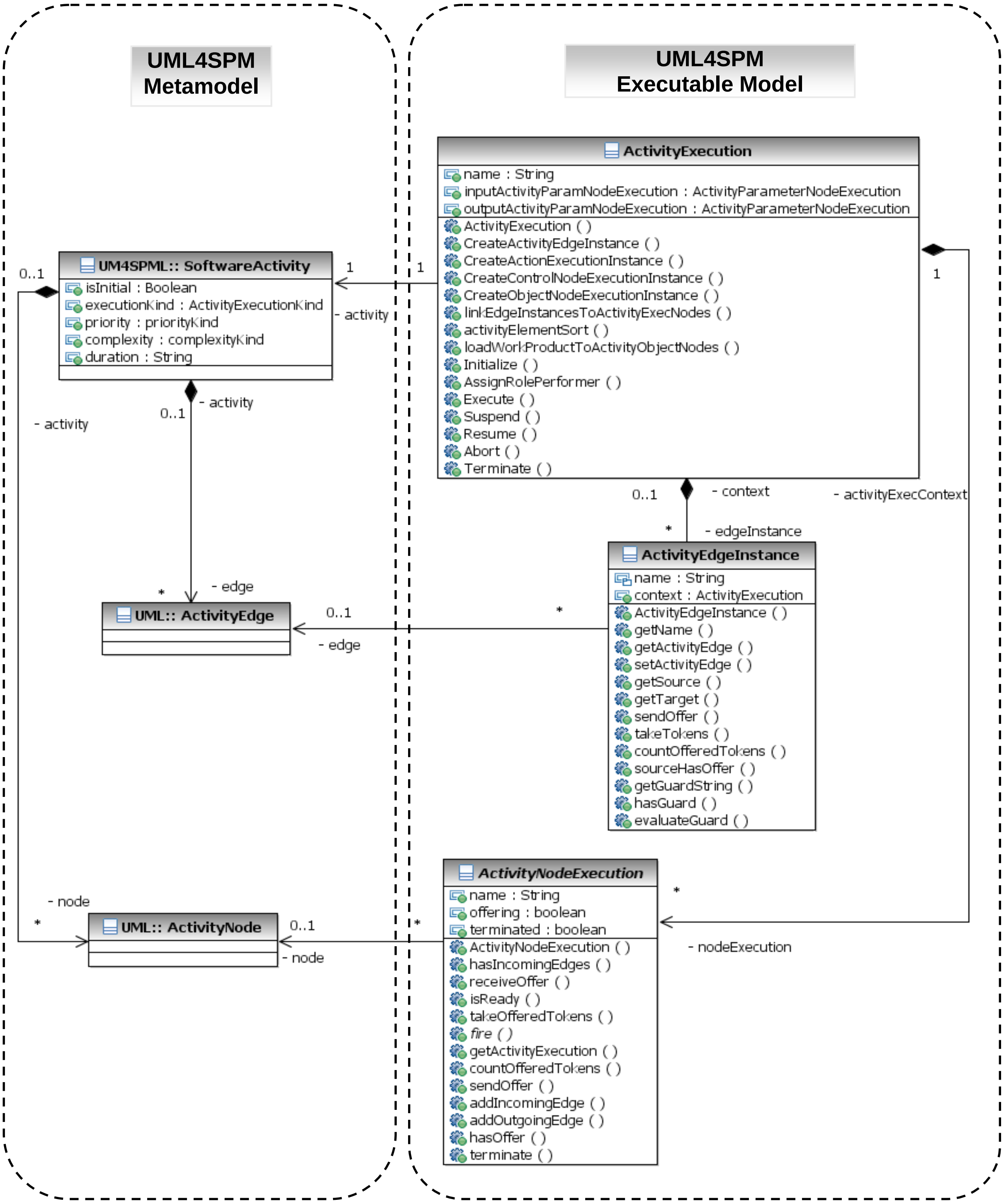


Figure 8.1. The Execution Model approach

In the following we start the presentation of the executable model by the *Process Model Execution* class.

3.2. Process Model Execution Class

The *Process Model Execution* class represents the context of the process and a kind of container of all process's *Activity Executions*, *WorkProducts* and *Responsible Roles* (see figure 8.2). It is a sort of "main" of the process execution and it is not represented in the UML4SPM metamodel. This execution class has these main objectives:

- Loading of the UML4SPM software process model to be executed. The location of the process model to execute can be given in an interactive way or stored in a configuration file. The loading of the process model consists in representing the process model in terms of objects in memory;

- Once the process model loaded, get the list of the process's *Software Activities*. Check whether there is one and only one *Software Activity* with its *isInitial* property set at *true*, otherwise the process can't execute (can't know which one is the initial among all the activities). If the process model contains only one *Software Activity* and even if its *isInitial* property is not specified, it will launch its execution;

- Creating for each *Software Activity* in the process model, an *Activity Execution* instance;

- Calling the *Initialize ()* operation on all *Activity Execution instances* in order to instantiate *Activity Execution Contents* (*ActivityEdgeInstances* and *ActivityNodeExecutions*) and to prepare the activity to be executed;

- Once *Activity Executions* initialized, look for *Role Performers* susceptible to take in charge the performing of these activities. *Role Performers* are selected by matching their *skills* with *Responsible Roles* required *qualifications*;

- Once *Role Performers* assigned to *Activity Executions*, start the initial *Activity Execution* by calling its *Execute ()* operation.

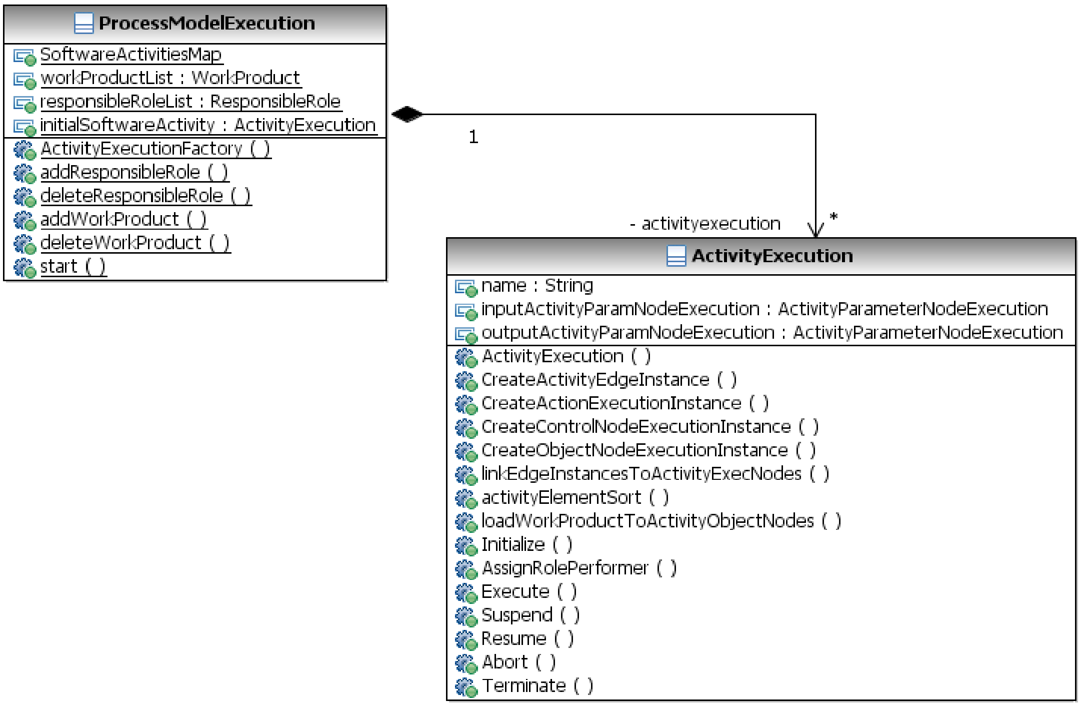

ProcessModelExecution
SoftwareActivitiesMap
workProductList : WorkProduct
responsibleRoleList : ResponsibleRole
initialSoftwareActivity : ActivityExecution
ActivityExecutionFactory ( )
addResponsibleRole ( )
deleteResponsibleRole ( )
addWorkProduct ( )
deleteWorkProduct ( )
start ( )
1
- activityexecution
*
ActivityExecution
name : String
inputActivityParamNodeExecution : ActivityParameterNodeExecution
outputActivityParamNodeExecution : ActivityParameterNodeExecution
ActivityExecution ( )
CreateActivityEdgeInstance ( )
CreateActionExecutionInstance ( )
CreateControlNodeExecutionInstance ( )
CreateObjectNodeExecutionInstance ( )
linkEdgeInstancesToActivityExecNodes ( )
activityElementSort ( )
loadWorkProductToActivityObjectNodes ( )
Initialize ( )
AssignRolePerformer ( )
Execute ( )
Suspend ( )
Resume ( )
Abort ( )
Terminate ( )

Figure 8.2. The Process Execution and Activity Execution Classes

Coupling of UML4SPM Process Models and their Execution Class Instances

The *Process Execution* class keeps a trace of all its contents (*Activity Executions, WorkProducts, Responsible Roles*, etc.) and of all the mappings between the UML4SPM process model elements and all their corresponding execution classes. In addition, all instances of the UML4SPM execution model have a reference to their corresponding elements in the UML4SPM process model loaded in memory. Thus, at runtime, when it is necessary, instances of execution classes extract data from process model elements. The data is not duplicated within the execution classes. This strong (coupling) relationship between the process model and its execution model makes that execution instances are always up-to-date with their corresponding process elements in case of these are modified.

The *Process Execution* class is proper to the UML4SPM execution model. Its Java implementation is given in the Appendix C of this document. In the next section, we present the *Activity Execution* class, which represents the execution behavior of UML4SPM *Software Activities*.

3.3. Activity Execution Class

An *Activity Execution* represents the execution behavior of a *Software Activity*. Before to launch the *Activity Execution* by calling its *Execute ()* operation, the *Activity Execution* instance calls the set of operations required for creating the *ActivityEdgeInstances* and *ActivityNodeExecutions* (e.g., *ControlNodeExecution* instances, *ObjectNodeExecution* instances, *ActionExecution* instances, etc) it contains. These operations are called from the *Initialize()* operation. Thus, the activity execution creates as many *ActivityEdgeInstance* instances and *ActivityNodeExecution* instances as *ActivityEdge* and *ActivityNode* instances, respectively, owned by the *Software Activity* corresponding to the *Activity Execution*.

Once the *Activity Execution* contents created, the *AssignRolePerformer()* operation is called in order to assign a role performer to the activity. The role performer is selected according to the "*qualifications*" described in the *Responsible Role* element defined in the process model and which is in charge of the *Software Activity*. When a role performer's (e.g., an Agent) *skills* match the *qualifications* of the activity's *Responsible Role* and its state is "available", the agent is selected as a potential performer of the activity. It is required that only one agent accepts to take in charge the realization of the activity in order to start the execution of the activity (call the *Execute()* operation).

Activity Execution contains also operations for suspending or resuming the execution of the activity (i.e., *Suspend()* and *Resume()*), for aborting the activity (i.e., *Abort()*) and finally terminating the activity (i.e., *terminate()*).

The execution of an activity is effectively realized by the execution of its activity node executions, therefore the *Execute()* operation of the class *Activity Execution* triggers the execution of the activity by concurrently calling the *receiveOffer()* operation on all of its *ActivityNodeExecution* instances that correspond to activity nodes with no incoming edges (nodes that need not to wait for any input to begin executing) and by putting a control token on all activity's *Initial Nodes*. The details of how activity nodes are executed are discussed in the following subsections and depend ultimately on the kind of the activity node to execute (*ActivityNodeExecution* is an abstract class having as subclasses, *ControlNodeExecution*, *ObjectNodeExecution*, and *ActionExecution*).

Execution of an activity terminates when all its node executions are terminated or when the execution of an ActivityFinalNode (by an ActivityFinalNodeExecution) completes.

Comparing with the *Activity Execution* class defined in the *Executable UML* specification, the one we defined in the context of UML4SPM execution model has eleven operations more. These operations deal principally with the initialization of the activity execution (e.g., *CreateActivityEdgeInstance()*, *CreateControlNodeExecutionInstance()*, etc.) but also with the control aspects of the activity execution (e.g., *Suspend()*, *Resume()*, *AssignRolePerformer()*, etc.). We implemented these operations in Java. The code of the *Activity Execution*'s operations is given in the Appendix C of this document.

For the execution of a *Software Activity*, the definition of an *Activity Execution* is not enough. We still need to define execution classes of its contents. As we know, a UML4SPM *Software Activity* inherits a UML2.0 *Activity*. At the higher level, a UML2.0 *Activity* is composed of *Activity Edges* (i.e., *Control Flows* and *Object Flows*) and *Activity Nodes* (i.e., *Control Nodes*, *Object Nodes* and *Actions*). Thus, we need to define the execution classes of Activity's contents. In the following, we present the *ActivityEdgeInstance* and *ActivityNodeExecution* classes which represent the building bloc of the *Activity Execution*. Then, we will go through the presentation of their subclasses.

3.4. ActivityEdgeInstance and ActivityNodeExecution Classes

The execution of an *Activity* is carried out by the execution of its *ActivityEdgeInstances* and *ActivityNodeExecutions*. They represent the pivotal classes of the execution model (see figure 8.3.). More particularly, the *ActivityNodeExecutions* class, which capitalizes/generalizes the execution behavior of all its subclasses (i.e., *ObjectNodeExecution*, *ControlNodeExecution* and *ActionExecution*) which in their turns have subclasses (e.g., *DecisionNodeExecution*, *InputPinExecution*, *CallBehaviorActionExecution*, etc). The semantics of the execution behavior respects the one given by the UML2.0 standard. Once these execution classes defined, their subclasses can directly inherit this execution behavior and then only few operations have to be redefined in order to take into account the execution behavior proper to the subclass. Examples of such operations are the *fire()* operation (in case of *ObjectNodeExecution* and *ControlNodeExecution*) and *doAction()* operation (in case of *Action Execution* subclasses (*CallbehaviorActionExecution*, *CallOperationActionExecution*, etc.).

ActivityEdgeInstances and *ActivityNodeExecutions* being so important, in the following we describe them in details. At the end of this subsection, a UML2.0 sequence diagram will sum up the set of operations involved in the execution behavior of these classes and when they are triggered. In the Appendix C, we provide the Java implementation of these classes and their operations.

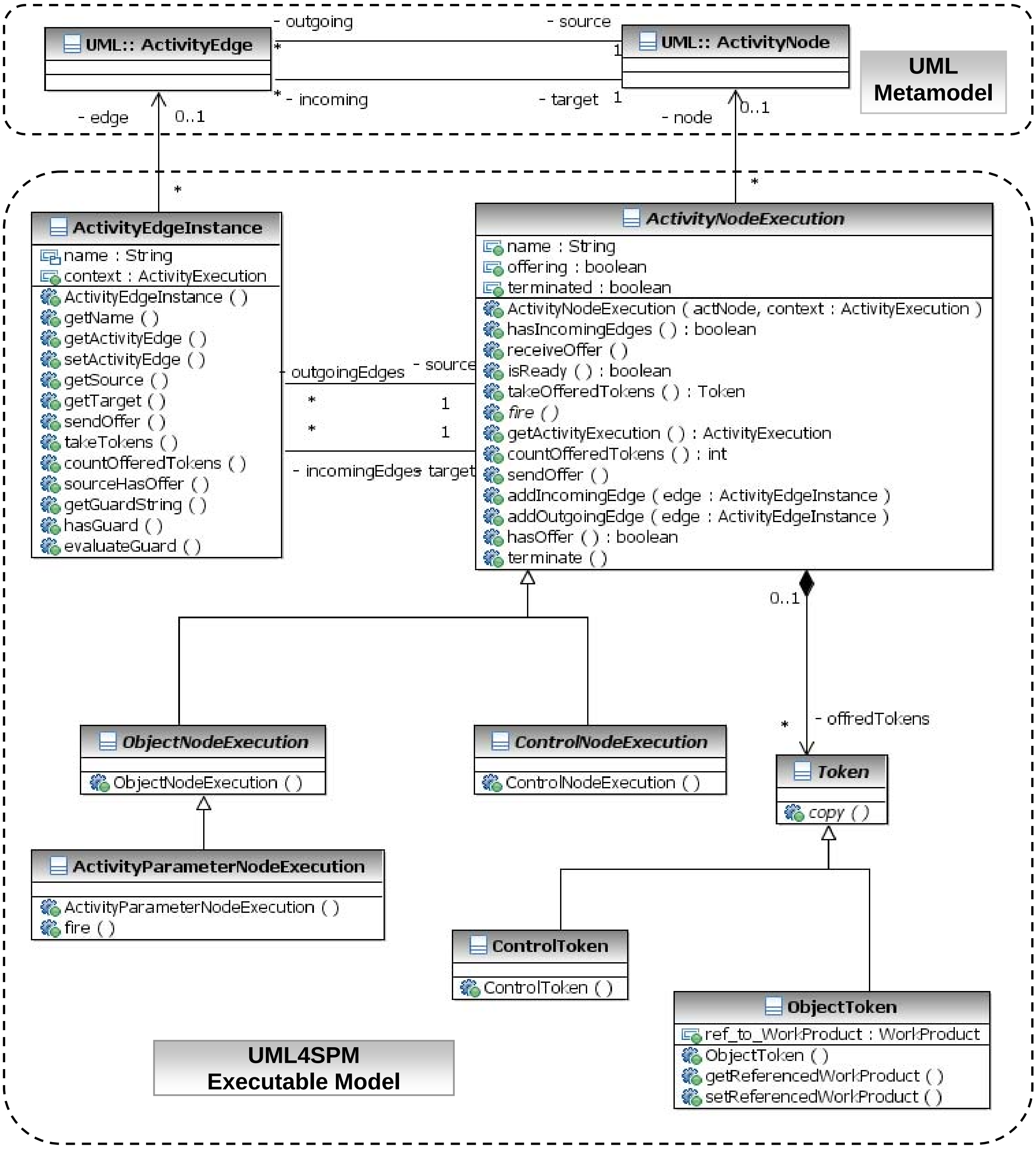


Figure 8.3. The Building Blocks of Activity Execution: The *ActivityEdgeInstance* and *ActivityNodeExecution* classes.

Execution Behavior

In UML2.0, the execution semantics of activities is based on *token flows*. By *flow*, we mean that the execution of one node affects, and is affected by, the execution of other nodes, and such dependencies are represented by *edges* in the activity diagram. A *token* contains an object, datum, or locus of control, and is present in the activity diagram at a particular node. Each token is distinct from any other, even if it contains the same value as another. The UML2.0 standard does not concretely define a Token metaclass however; this notion is only used to express the semantics of activities.

In the UML4SPM execution model, we defined the token class and we differentiate between two kinds of tokens. *Control* tokens and *Object* tokens. When an *Action Execution* completes, it creates a control token and offers it to its all outgoing *ActivityEdgeInstances*. Also, if an *Activity Execution* contains an *InitialNodeExecution*, at *initialization* phase of the *Activity Execution*, a control token is created and placed in the initial node. Object tokens are exchanged between object nodes (*Input* and *Output Pins* of actions, *Data Store Nodes*, etc.) and may traverse control nodes. For instance, when an *Action Execution* completes and if it provides an output, an object token with a reference to the *OutputPinExecution* type is created and placed in the *OutputPinExecution* instance of the *Action Execution*. In the context of UML4SPM, an *OutputPinExecution* can only be typed by *WorkProducts* or subclasses of the *WorkProduct* metaclass.

ActivityNodeExecutions and *ActivityEdgeInstances* follow token flow rules as defined by the UML2.0 standard. *ActivityNodeExecutions* control when tokens enter or leave them. *ActivityEdgeInstances* have rules about when a token may be taken from the source *ActivityNodeExecution* and moved to the target *ActivityNodeExecution*. A token traverses an *ActivityEdgeInstance* when it satisfies the rules for target *ActivityNodeExecution*, *ActivityEdgeInstance*, and source *ActivityNodeExecution*, all at once. This means that a source *ActivityNodeExecution* can only offer tokens to the outgoing *ActivityEdgeInstances*, rather than force them along the *ActivityEdgeInstance*, because the tokens may be rejected by the *ActivityEdgeInstance* or the target *ActivityNodeExecution* on the other side.

The *ActivityEdgeInstance* acts as a mediator between its source *ActivityNodeExecution* offering tokens and its target *ActivityNodeExecution* taking tokens. Tokens are effectively held by the offering *ActivityNodeExecution* until the receiving one is ready to take them. As such mediator, an *ActivityEdgeInstance* provides the following functionality: checks whether its source is offering any token, send offers of tokens from its source to its target and take the offered tokens from its source to its target *ActivityNodeExecution* (see figure 8.4).

The execution of an *ActivityNodeExecution* instance begins when its *receiveOffer()* operation is called (either by its containing *Activity Execution* or by an incoming *ActivityEdgeInstance*). This causes a call to *isReady()* operation on itself to check whether the execution can proceed. This is true if the *ActivityNodeExecution* has no incoming edges. Otherwise, it calls *sourceHasOffer()* operation on all its incoming *ActivityEdgeInstance*. In turn, this causes the *ActivityEdgeInstance* to call *hasOffer()* operation on its source *ActivityNodeExecution* to check whether it is still making an offer. If this is the case, returns of *hasOffer(), sourceHasOffer()* and *isReady()* will be a Boolean value of true. Then the *ActivityNodeExecution* calls the *fire()* operation on itself. This is an abstract operation whose method is found in each of the concrete *ActivityNodeExecution* and *ActionExecutions*. If, in order to execute, the *ActivityNodeExecution* needs to take tokens from its incoming edge it calls *takeTokens()* on its incoming edge, which in turn calls *takeOfferedTokens()* on its source *ActivityNodeExecution*. This causes the removal of all *offeredTokens* and the setting of the *offering* attribute to *false* (so that the source *ActivityNodeExecution* is no longer holding any token and hence no longer making an offer).

Tokens will be consumed by the executing *ActivityNodeExecution* accordingly depending on its type and, eventually, as a result of executing the *fire()* operation, tokens may be produced and written to the *offeredTokens* of the executing

ActivityNodeExecution (where they will be held up to its consumption), which also sets its *offering* attribute to *true* (to indicate it is now making an offer) and then concurrently calls *sendOffer()* on all its outgoing edges (and, consequently, this will cause each outgoing *ActivityEdgeInstance* to call *receiveOffer()* on its target *ActivityNodeExecution*).

The sequence diagram given in figure 8.4 summarizes the sequence of operation calls aiming at preparing the *ActivityNodeExecution* to execute its behavior. More details on how we implemented these operations can be found in the Appendix of this document

Comparing to the execution model proposed by the *Executable UML* initiative, the *ActivityEdgeInstance* class we defined is enriched with new operations. These operations deal with guard evaluations, an important aspect which was not taken into account in the *Executable UML* specification. The expression of guards is important in the context of software process modeling, since they drive the process workflow. In the UML4SPM execution model, we take in charge the evaluation of guards expressions thanks to the *hasGuard()* and *evaluateGuard()* operations (figure 8.4). Guard can be expressed upon any *Process Element's property* (*Software Activity*, *WorkProduct* and *Responsible Role*) of the process model. The syntax of the guard expression that have to be specified while editing UMl4SPM process models is: `"ProcessElementKind.ProcessElementName.PropertyName=Value"`. As an Example (cf. Chapter 7, section 6.2), the following guard expression upon a WorkProduct called UMLAnalysisModel: `WorkProduct.UMLAnalysisModel.state=checked`. In the current implantation of our process engine, only guard expressions on *WorkProduct's* properties are taken into account.

In the near future, we envisage to use OCL as language for expressing more sophisticated guards upon elements of the process model and to integrate an OCL checker to our process engine.

In the following subsections, we present the execution behavior of the *ActivityNodeExecution* subclasses i.e., *ControlNodeExecution*, *ObjectNodeExecution* and *ActionExecution*. We start by the *ControlNodeExecution*.

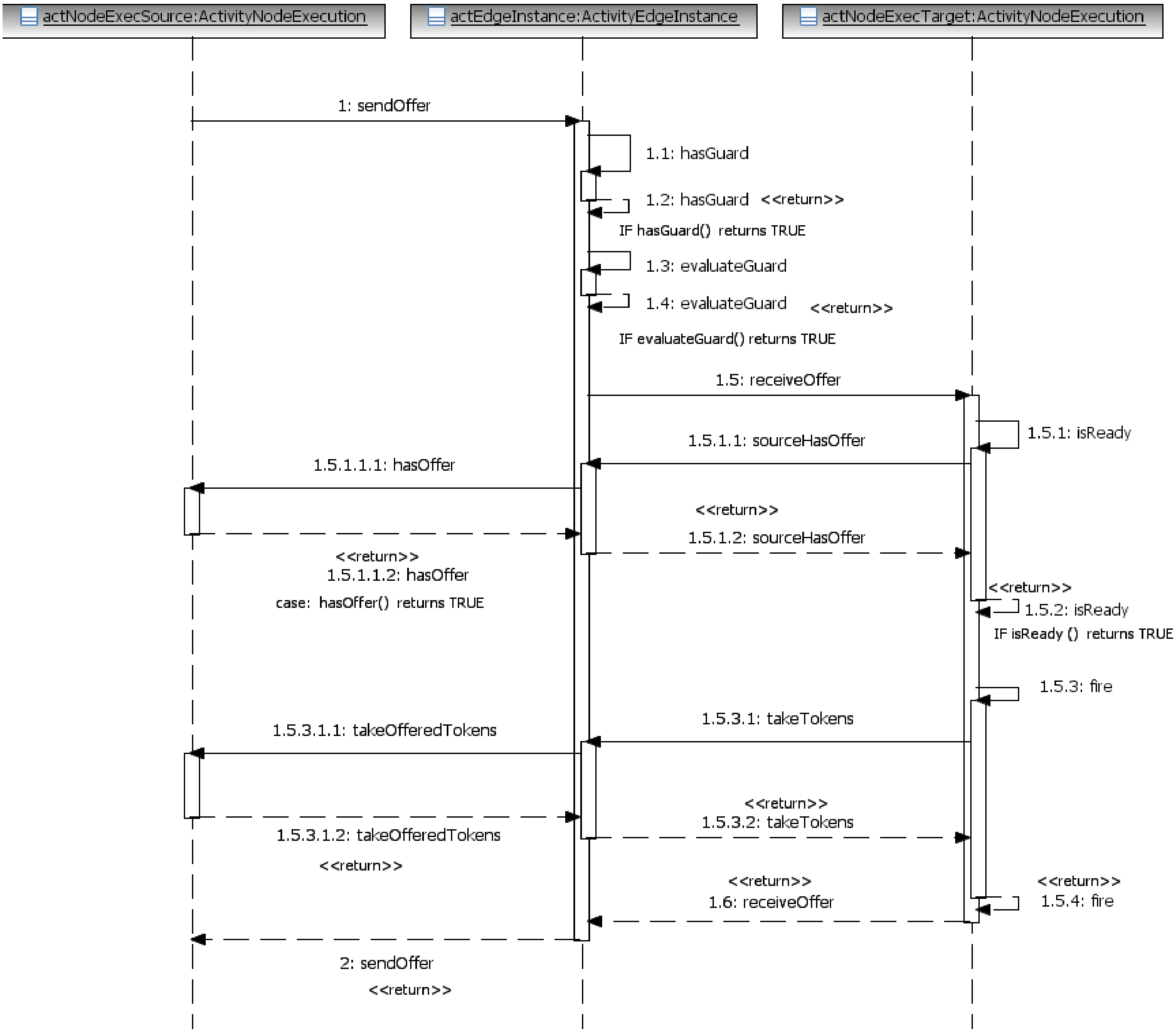


Figure 8.4. ActivityNodeExecution and ActivityEdgeInstance interactions

3.5. ControlNodeExecution Class

The *ControlNodeExecution* abstract class represents the execution behavior of UML2.0 *Control Nodes*. The general behavior is inherited from the *ActivityNodeExecution* class and then is refined by redefining the *fire()* operation principally and other operations such as *hasOffer()*, *receiveOffer()*, etc. within each of the *ControlNodeExecution* subclasses i.e. *InitialNodeExecution*, *DecisionNodeExecution*, *ForkNodeExecution*, *MergeNodeExecution*, *ActivityFinalNodeExecution*, and *JoinNodeExecution* (see figure 8.5).

ControlNodeExecutions act as traffic switches (i.e., Tokens cannot "rest" at *ControlNodeExecutions*) managing tokens as they make their way between *ObjectNodeExecutions* and *ActionExecutions*, which are the nodes where tokens can rest for a period of time. *InitialNodeExecutions* are exceptions from this rule [OMG 07b].

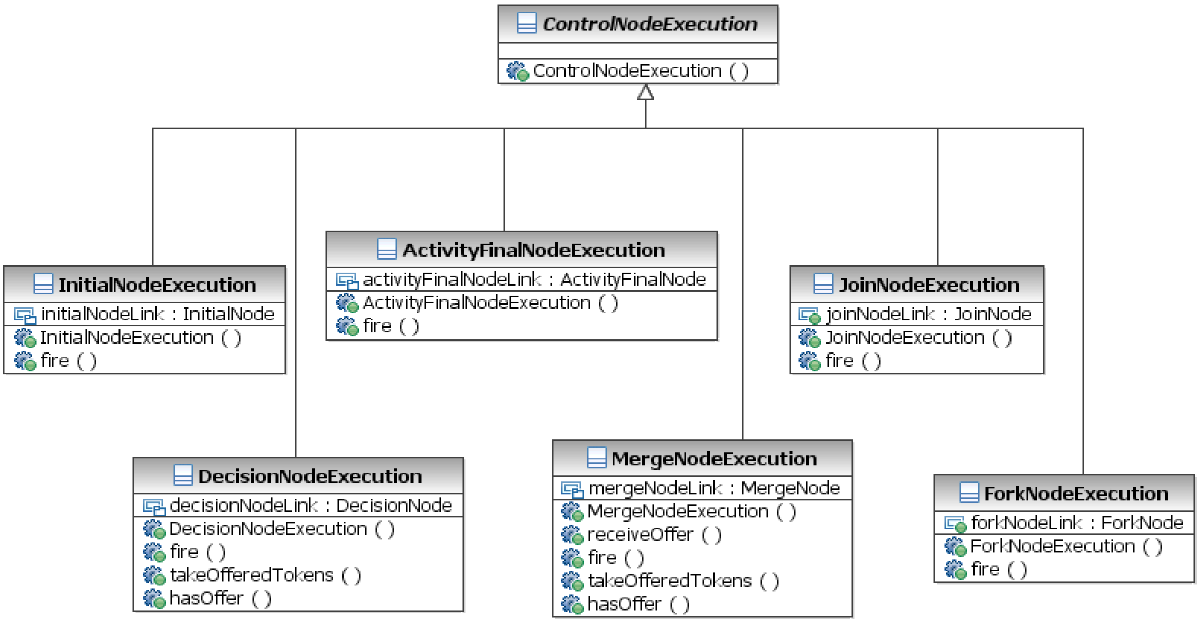


Figure 8.5. ControlNodeExecution Abstract Class and its Concrete Execution Subclasses.

The firing of an *InitialNodeExecution* produces a control token which is then offered on all its outgoing *ActivityEdgeInstances*. The execution of an *ActivityFinalNode* involves taking all tokens from all its incoming edges and call the *terminate()* operation on the *Activity Execution* directly containing the executing *ActivityFinalNodeExecution*.

The execution of a *JoinNode* is completed by taking tokens offered by all of its incoming edges and then offering them. *JoinNodeExecution* redefines the *isReady()* operation to check that all the source *ActivityNodeExecution* instances which are source of its incoming edges are offering tokens.

The execution of a *MergeNodeExecution* is completed by taking tokens offered by any incoming *ActivityEdgeInstance* whose source *ActivityNodeExecution* has completed and offer them. In the UML4SPM execution model, in order to specify the *MergeNodeExecution* behavior we redefined the *receiveOffer()*, *fire()*, *takeOfferedTokens()* and *hasOffer()* operations. This is due to the fact that as we said earlier, *Control Nodes* act as traffic switches and then, they simply have to forward the offers they receive or confirmations of offers. The same think is done regarding tokens; they are simply transferred form the source *ActivityNodeExecution* instance to the target *ActivityNodeExecution* instance without "resting" in the *MergeNodeExecution*. Finally, the call of the *isReady()* operation on this execution node returns always true. The Java code implementing this execution semantics is given in Appendix C. Each line of code is documented in natural language. Of course only the redefined operations are presented. For the operations and properties inherited from the *ActivityNodeExecution*, please see the code of the *ActivityNodeExecution* class (also given in Appendix C).

The *Executable UML* specification does not define an execution class for the UML2.0 *Decision Node* element, claming that it can be replaced by the *Conditional Node* element. In UML4SPM process models, we make an extensive use of *Decision Nodes* since they are used to drive the process work flow. This can be done automatically under the condition that guards are specified on outgoing edges. In the absence of guards on outgoing edges, the *DecisionNodeExecution* interacts with the process modeler or agent in order to ask him/her to choose between one of the outgoing *ActivityEdgeInstances*.

The Java code describing The *DecisionNode* behavior is given in Appendix C. That latter is described within the *fire()* operation. The *hasOffer()* and *takeOfferedTokens()* operations were also redefined so that, as in the case of *MergeNodeExecution*, it just forwards the request asking if the node is making an offer to the source *ActivityNodeExecution* and to take tokens from the source *ActivityNodeExecution*. For the operations and properties inherited from the *ActivityNodeExecution*, please see the code of the *ActivityNodeExecution* class (see Appendix C).

Finally, when writing this document the behavior implementation of the *ForkNodeExecutioni* was underway. Its execution semantics can be found in the UML2.0 specification [OMG 07b].

In the next subsection, we present the *ObjectNodeExecution* class and its subclasses *PinExecution (InputPinExecution* and *OutputPinExecution)* and *ActivityParameterNodeExecution*.

3.6. ObjectNodeExecution Class

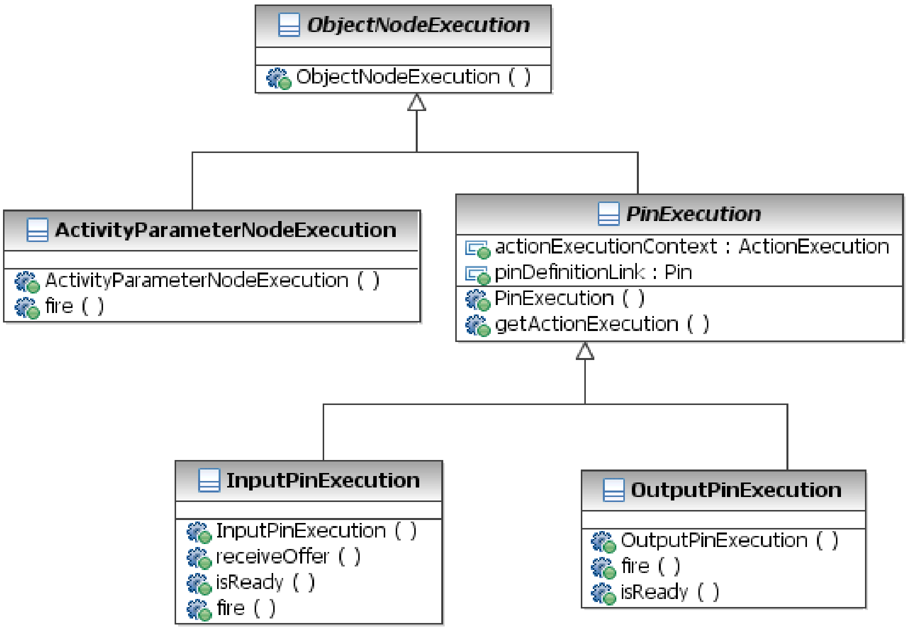


Figure 8.6. ObjectNodeExecution and its subclasses.

As any *Activity Node*, *Object Nodes* are live objects and have a behavior which is represented at runtime by an *ObjectNodeExecution* (see figure 8.6). The *ObjectNodeExecution* class is an abstract class, which inherits its general behavior from the *ActivityNodeExecution* class. The behavior of its subclasses is then described by implementing the *fire()* operation.

InputPinExecution is an *ObjectNodeExecution* where *Object Tokens* can rest before to be consumed by the *Action Execution* owning the *InputPinExecution*. Thus, when an *InputPinExecution* receives an offer (i.e., a call of the *receiveOffer()* on the *InputPinExecution*) it just forwards the offer to its *Action Execution*. If the *Action Execution* is ready, then tokens are taken from the source *ActivityNodeExecution* making an offer to the *InputPinExecution* and added to the *InputPinExecution OfferedToken* list. This done by calling the *fire()* operation on the *InputPinExecution* (see the Java implementation of the *fire()* on Appendix C).

When the *Action Execution* behavior is fired, *Object Tokens* are removed form the *OfferedToken* list of the *InputPinExecution*, consumed by the action, and if any *Object Tokens* are to be produced by the *Action Execution*, they are transferred to its *OutputPinExecutions*. The *isReady ()* operation is redefined in the context the *OutputPinExecution* class to always return true.

Finally, *ActivityParameterNodeExecutions (APNE)* are to represent the execution behavior of the UML2.0 *Activity Parameter Node* metaclass. *Activity parameter nodes* are object nodes at the beginning and end of flows that provide a means to accept inputs to an activity and provide outputs from the activity, through the activity parameters.

The *ActivityParameterNodeExecution* implements the *fire()* operation depending on whether it is an input *APNE* (if it has no incoming edges) or an output *APNE* (if it has no outgoing edges) ActivityParameterNode. In the first case, it takes the value of its corresponding input parameter and offers it as an object token. In the latter case, it takes the tokens offered by the source *ActivityExecutionNode* instances of its incoming edges and offers them as object tokens. The Java implementation of the APNE class is given in the Appendix C of this document.

In the UML4SPM execution model, we assume that the *multiplicity* and *upper bound* properties of the *PinExecution* and *APNE* are always equal to one.

3.7. ActionExecution Class

Figure 8.7 lists the set of *Action Execution* classes taken into account by the UML4SPM execution model. The abstract *Action Execution* class capitalizes the set of operations that have to be invoked by each of its subclasses whatever its kind. Examples of such operations are the *CreateActionInputPinExecution()* and *CreateActionOutputPinExecution()* operations which aim at creating the *PinExecution* instances of the *Action Execution*.

The *Action Execution's isReady()* operation is redefined in order to check, in addition to that sources of all its incoming edges are still making an offer, that its *InputPinExecutions* are ready. This done by calling from the *isReady()* operation, the *isInputPinExecutionReady()* operation.

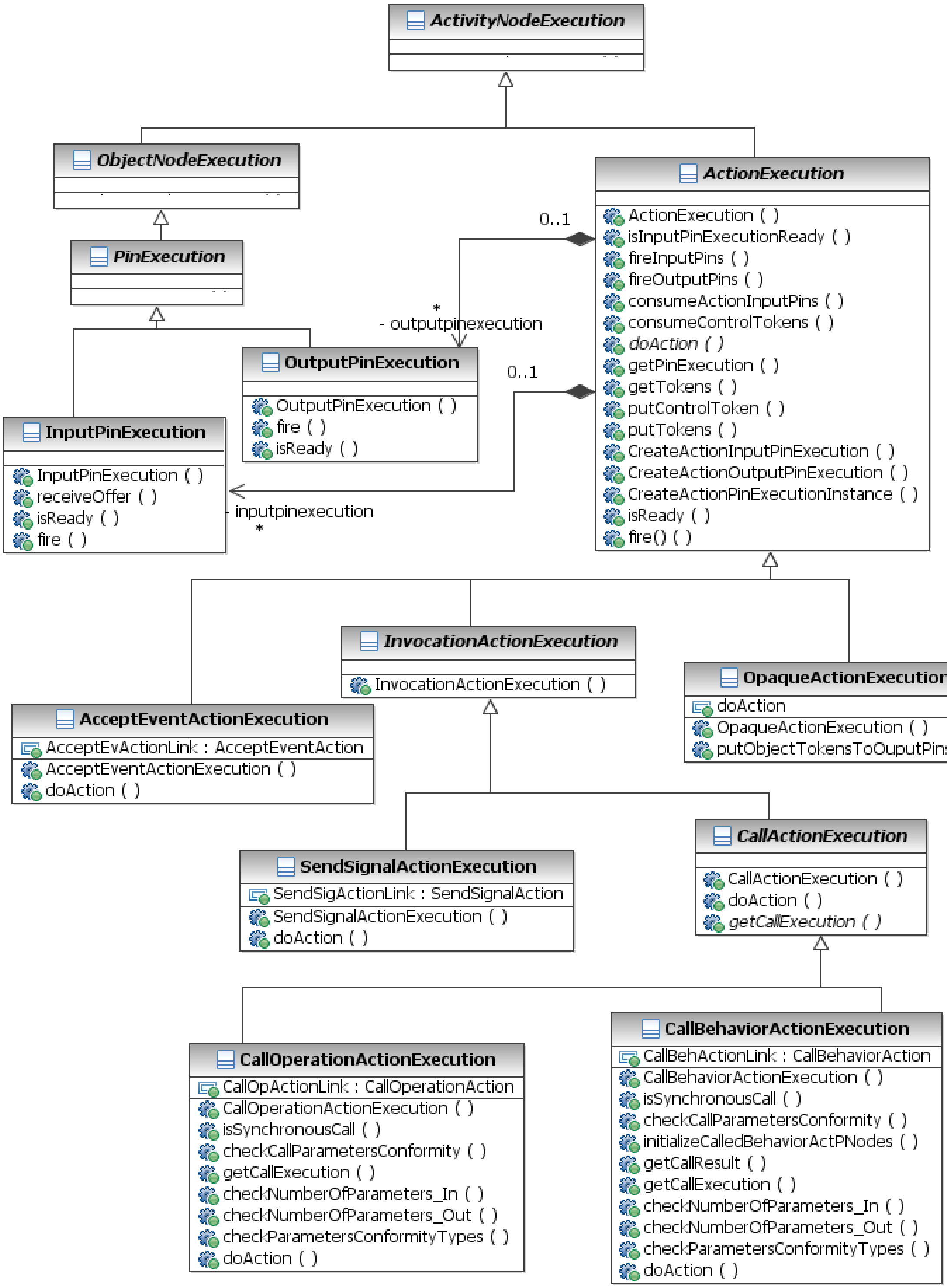

ActivityNodeExecution
ObjectNodeExecution
PinExecution
ActionExecution
ActionExecution ( )
isInputPinExecutionReady ( )
fireInputPins ( )
fireOutputPins ( )
consumeActionInputPins ( )
consumeControlTokens ( )
doAction ( )
getPinExecution ( )
getTokens ( )
putControlToken ( )
putTokens ( )
CreateActionInputPinExecution ( )
CreateActionOutputPinExecution ( )
CreateActionPinExecutionInstance ( )
isReady ( )
fire() ( )
0..1
*
- outputpinexecution
0..1
OutputPinExecution
OutputPinExecution ( )
fire ( )
isReady ( )
InputPinExecution
InputPinExecution ( )
receiveOffer ( )
isReady ( )
fire ( )
- inputpinexecution
*
InvocationActionExecution
InvocationActionExecution ( )
OpaqueActionExecution
doAction
OpaqueActionExecution ( )
putObjectTokensToOuputPins ( )
AcceptEventActionExecution
AcceptEvActionLink : AcceptEventAction
AcceptEventActionExecution ( )
doAction ( )
CallActionExecution
CallActionExecution ( )
doAction ( )
getCallExecution ( )
SendSignalActionExecution
SendSigActionLink : SendSignalAction
SendSignalActionExecution ( )
doAction ( )
CallBehaviorActionExecution
CallBehActionLink : CallBehaviorAction
CallBehaviorActionExecution ( )
isSynchronousCall ( )
checkCallParametersConformity ( )
initializeCalledBehaviorActPNodes ( )
getCallResult ( )
getCallExecution ( )
checkNumberOfParameters_In ( )
checkNumberOfParameters_Out ( )
checkParametersConformityTypes ( )
doAction ( )
CallOperationActionExecution
CallOpActionLink : CallOperationAction
CallOperationActionExecution ( )
isSynchronousCall ( )
checkCallParametersConformity ( )
getCallExecution ( )
checkNumberOfParameters_In ( )
checkNumberOfParameters_Out ( )
checkParametersConformityTypes ( )
doAction ( )

Figure 8.7. UML4SPM Action Execution Classes

Once the *Action Execution* is ready to execute, it calls the *fire()* operation. This will in first step trigger the firing (by calling the *fireInputPins ()* operation) of the action's *InputPinExecutions*. This will cause the transfer of *Object Tokens* from source *ActivityNodeExecution* to the action's *InputPinExecution* instances. In a second step, *Control Tokens* on action's incoming edges are consumed by calling the *consumeControlTokens()* operation. Then, the *Action Execution* calls the *doAction()* operation on it. This operation is to be defined on each subclass of the *Action Execution* class. It represents the behavior proper to the kind of action. We will see an example in the case of the *CallBehaviorExecutionAction's doAction()* operation. In figure 8.8 we give a sequence diagram that summarizes the general *Action Execution* behavior.

Comparing with the *Executable UML specification* [OMG 06e], the *Action Execution* class we propose identifies nine operations more. These operations deal with action's *PinExecution* creations, the firing of *InputPinExecutions* and *OutputPinExecutions*, and token consumptions/creations. The Java implementation we propose for this class is given in the Appendix C of this document.

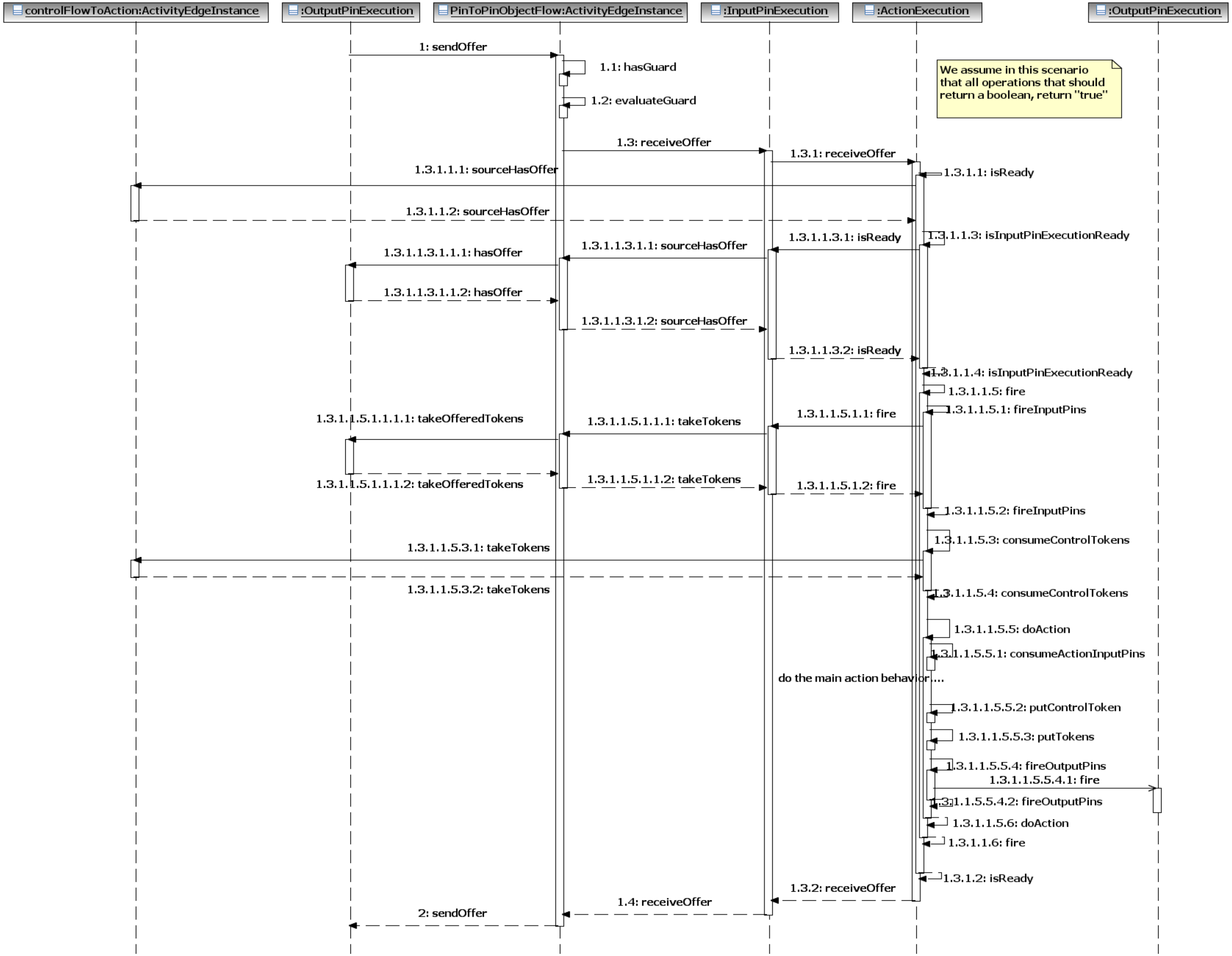

controlFlowToAction:ActivityEdgeInstance
:OutputPinExecution
PinToPinObjectFlow:ActivityEdgeInstance
:InputPinExecution
:ActionExecution
:OutputPinExecution
1: sendOffer
1.1: hasGuard
1.2: evaluateGuard
We assume in this scenario that all operations that should return a boolean, return "true"
1.3: receiveOffer
1.3.1: receiveOffer
1.3.1.1.1: sourceHasOffer
1.3.1.1: isReady
1.3.1.1.2: sourceHasOffer
1.3.1.1.3.1: isReady
1.3.1.1.3: isInputPinExecutionReady
1.3.1.1.3.1.1: sourceHasOffer
1.3.1.1.3.1.1.1: hasOffer
1.3.1.1.3.1.1.2: hasOffer
1.3.1.1.3.1.2: sourceHasOffer
1.3.1.1.3.2: isReady
1.3.1.1.4: isInputPinExecutionReady
1.3.1.1.5: fire
1.3.1.1.5.1: fireInputPins
1.3.1.1.5.1.1: fire
1.3.1.1.5.1.1.1: takeOfferedTokens
1.3.1.1.5.1.1.1: takeTokens
1.3.1.1.5.1.1.2: takeOfferedTokens
1.3.1.1.5.1.1.2: takeTokens
1.3.1.1.5.1.2: fire
1.3.1.1.5.2: fireInputPins
1.3.1.1.5.3: consumeControlTokens
1.3.1.1.5.3.1: takeTokens
1.3.1.1.5.3.2: takeTokens
1.3.1.1.5.4: consumeControlTokens
1.3.1.1.5.5: doAction
1.3.1.1.5.5.1: consumeActionInputPins
do the main action behavior ...
1.3.1.1.5.5.2: putControlToken
1.3.1.1.5.5.3: putTokens
1.3.1.1.5.5.4: fireOutputPins
1.3.1.1.5.5.4.1: fire
1.3.1.1.5.5.4.2: fireOutputPins
1.3.1.1.5.6: doAction
1.3.1.1.6: fire
1.3.1.2: isReady
1.3.2: receiveOffer
1.4: receiveOffer
2: sendOffer

Figure 8.8. Action Execution Behavior Sequence Diagram

In the following, we address only the *CallBehaviorActionExecution* and *OpaqueActionExecution* classes (see figure 8.8), which represent the execution behavior of the UML2.0 *CallBehaviorAction* and *OpaqueAction*, respectively. These actions are very important since they are extensively used in UML4SPM process models.

Synchronous and Asynchronous Software Activity calls

The *CallBehaviorAction* is used within a *Software Activity* in order to trigger the behavior of another *Software Activity*. The call can be synchronous or asynchronous. In the first case, the *Software Activity* waits that the called *Software Activity* terminates in order to pursue its execution. In case of an asynchronous call, the *Software Activity* has not to wait for the called *Software Activity* to terminate and both software activities execute **concurrently**.

Once prepared and in order to execute, the *CallBehaviorActionExecution* starts by checking that parameters of the call match with the *Activity Parameter Nodes* of the called *Software Activity* in number and types. This is ensured by calling the *checkCallParametersConformity()* operation, which in its turn calls the *checkNumberOfParameters_In()*, *checkParametersConformityTypes()* and *checkNumberOfParameters_Out()* (in case of synchronous call) operations.

If the *checkCallParametersConformity()* returns true then, the *initializeCalledBehaviorActPNodes()* is called. This operation aims at initializing the *ActivityParameterNodeExecutions* of the called activity by putting *Object Tokens* on them. If the call is asynchronous, a new thread is created, so that the current activity does not have to wait for the called *Software Activity* to terminate. If the call is synchronous the *Software Activity* waits for the called activity to terminate before getting back the result by calling the *getCallResult()* operation. The *getCallResult()* operation takes care of transferring the result of the called activity to the OutputPinExecution instances of the *CallBehaviorExecutionAction*. These OutputPinExecutions are then fired by calling the *fireOutputPins()* operation. Finally, before terminating, the action put a *Control Token* on its *offeredTokens* list (call of the *putControlToken()* operation) and send an offer on all its outgoing edges by calling the *sendOffer()* operation.

All these steps are carried out within the *CallbehaviorActionExecution's doAction()* operation. We give the corresponding Java implementation of this operation in the Appendix C of this document.

Comparing to the *Executable UML specification*, we defined seven more operations for the *CallBehaviorActionExecution*. These operations mostly deal with call parameters conformity checking, the initialization of *Object Tokens* in and out (to) of the called *Software Activity* and for controlling if whether the call is synchronous or not.

Process Execution Interactions with external applications

The last action execution class we present in this subsection is the *OpaqueActionExecution* class. In the UML4SPM execution model, this action has an important place since it allows the biding of the process execution to any external business application. By business application, we mean any executable application whatever its purpose. It may be a Graphic User Interface (GUI) to interact with the agent. The GUI can inform the agent with steps he has to carry out in order to realize the action, a link to guidance giving some hints and best practices, action deadlines, *WorkProducts* inputs to the action, the set of output *WorkProducts* he has to realize, etc. The agent can also give the action's progress percentage through the GUI, which can be used by a process monitoring application in order to anticipate some unexpected delays.

The *OpaqueActionExecution* class can also be used to interact with company's business applications or workflows, which opens many perspectives such as calling distant applications, binding to databases, calling web services, and so on.

In order to allow this kind of flexibility, in the current implementation of the *OpaqueActionExecution* we provide the means to execute the (Java) code specified in the *body* property of the *OpaqueAction*. At runtime, the *doAction()* of the *OpaqueActionExecution* starts by extracting the Java code from the action's *body* property. Then, it creates a Java class with a method called *ExecuteBody()* having as a body, the Java code extracted from the action. Using the Java *Runtime* Interface, the class is compiled without interrupting the *OpaqueActionExectuion*. Finally, we use the Java reflect API in order to load the Java class we compiled and which contains the Java code held by the *OpaqueAction's body* property. Then, a call to the *ExecuteBody()* operation is performed from the *OpaqueActionExectuion's doAction* operation to execute the code. All this is performed while the process is still running.

The general behavior of the *OpaqueActionExecution* can then be specialized by more specific actions. Currently, we are working to specify three kinds of actions. The first one aims at specifying a standard GUI to be used in actions requiring human interactions. The second kind of action is to allow tool modeling service calls while using the Model Bus approach, a work done in our team at LIP6 in order to allow interoperability between modeling tools. Finally, an action execution that allows calling distant web services from the process execution.

The *Executable UML* specification does not propose any execution class for the *OpaqueAction*. In the Appendix C of this document, the reader can find the Java implementation we propose for the *OpaqueActionExecution*.

In the next section, we present the UML4SPM process engine, which is based on the UML4SPM execution model we introduced here.

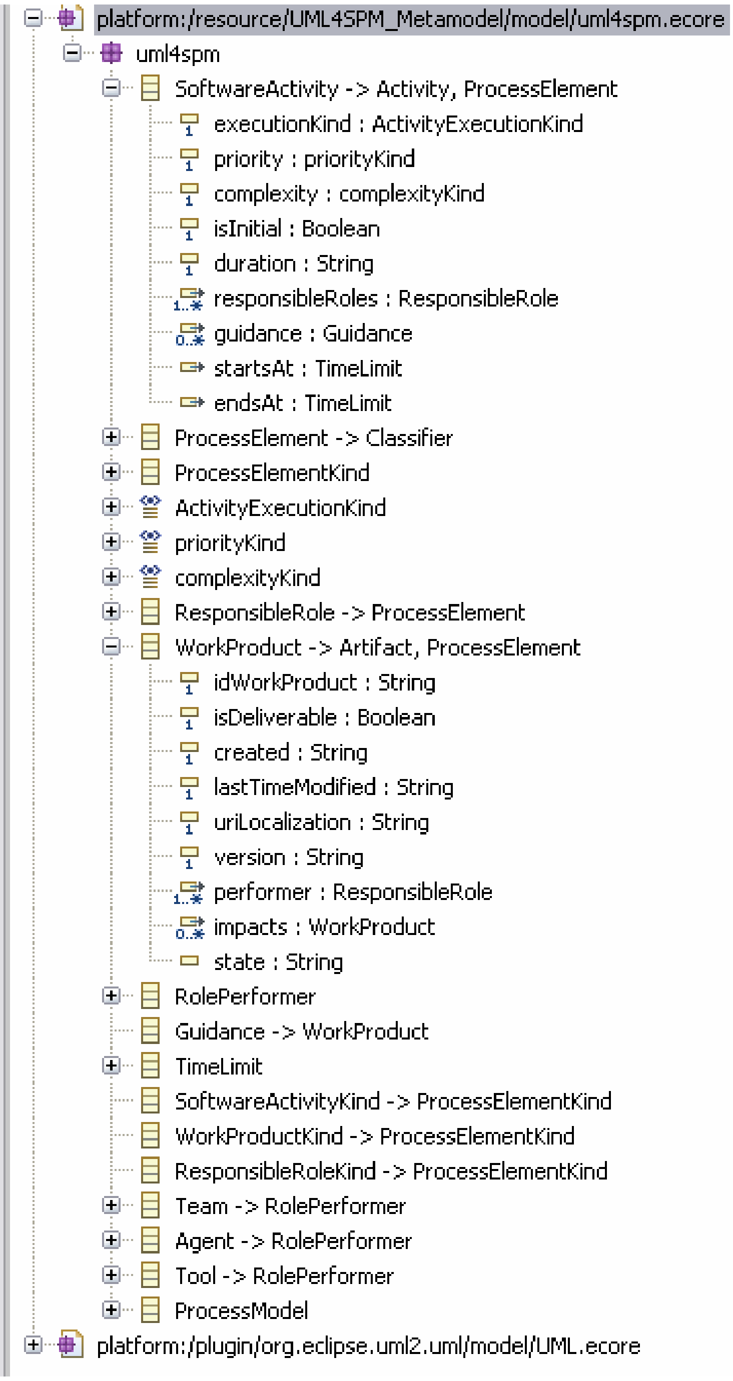
4. Execution of UML4SPM Process Models

Before presenting the UML4SPM process engine, we start by introducing the UML4SPM Process model Editor that will be used to produce process models input to the UML4SPM Process Execution Engine.

4.1. UML4SPM Process Model Editor

For the realization of the UML4SPM Process Model Editor we used the Eclipse Open Source Development Environment combined with the EMF and UML2.0 Plugins [eclipse].

Our first step for the realization of our editor was the definition of our metamodel. As we presented it in chapter 5, UML4SPM comes in form of MOF-compliant metamodel which extends some UML2.0 metaclasses. Figure 8.9 shows our metamodel defined within Eclipse using the EMF and UML2.0 APIs. We can notice for instance how the *Software Activity* metaclass inherits the UML2.0 *Activity* metaclass. Also, the *WorkProduct* metaclass which inherits the UML2.0 *Artifact* metaclass, and so on.



```
platform:/resource/UML4SPM_Metamodel/model/uml4spm.ecore
  uml4spm
    SoftwareActivity -> Activity, ProcessElement
      executionKind : ActivityExecutionKind
      priority : priorityKind
      complexity : complexityKind
      isInitial : Boolean
      duration : String
      responsibleRoles : ResponsibleRole
      guidance : Guidance
      startsAt : TimeLimit
      endsAt : TimeLimit
    ProcessElement -> Classifier
    ProcessElementKind
    ActivityExecutionKind
    priorityKind
    complexityKind
    ResponsibleRole -> ProcessElement
    WorkProduct -> Artifact, ProcessElement
      idWorkProduct : String
      isDeliverable : Boolean
      created : String
      lastTimeModified : String
      uriLocalization : String
      version : String
      performer : ResponsibleRole
      impacts : WorkProduct
      state : String
    RolePerformer
    Guidance -> WorkProduct
    TimeLimit
    SoftwareActivityKind -> ProcessElementKind
    WorkProductKind -> ProcessElementKind
    ResponsibleRoleKind -> ProcessElementKind
    Team -> RolePerformer
    Agent -> RolePerformer
    Tool -> RolePerformer
    ProcessModel
platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore
```

Figure 8.9. UML4SPM Metamodel Defined Within Eclipse

Once the metamodel defined, the UML4SPM Process Model Editor is generated on a simple click. It is now possible to define UML4SPM process models, which conform the UML4SPM metamodel. Process models will be stored using the OMG standard XMI format [OMG 05b]. Figure 8.10 gives an overview of the process model editor.

If the UML4SPM metamodel have to be modified, then the UML4SPM editor have to regenerated. This will not take more than few seconds. Additionally, if the modification is an extension to the metamodel (i.e., addition of a new attributes or metaclasses), the process models defined in a previous version can still be used within the new editor. If the modification to the UML4SPM is a suppression of a metaclass or attribute, then process models have to be migrated to the new version of the language though a model transformation. Model transformations are out of the scope of this document.

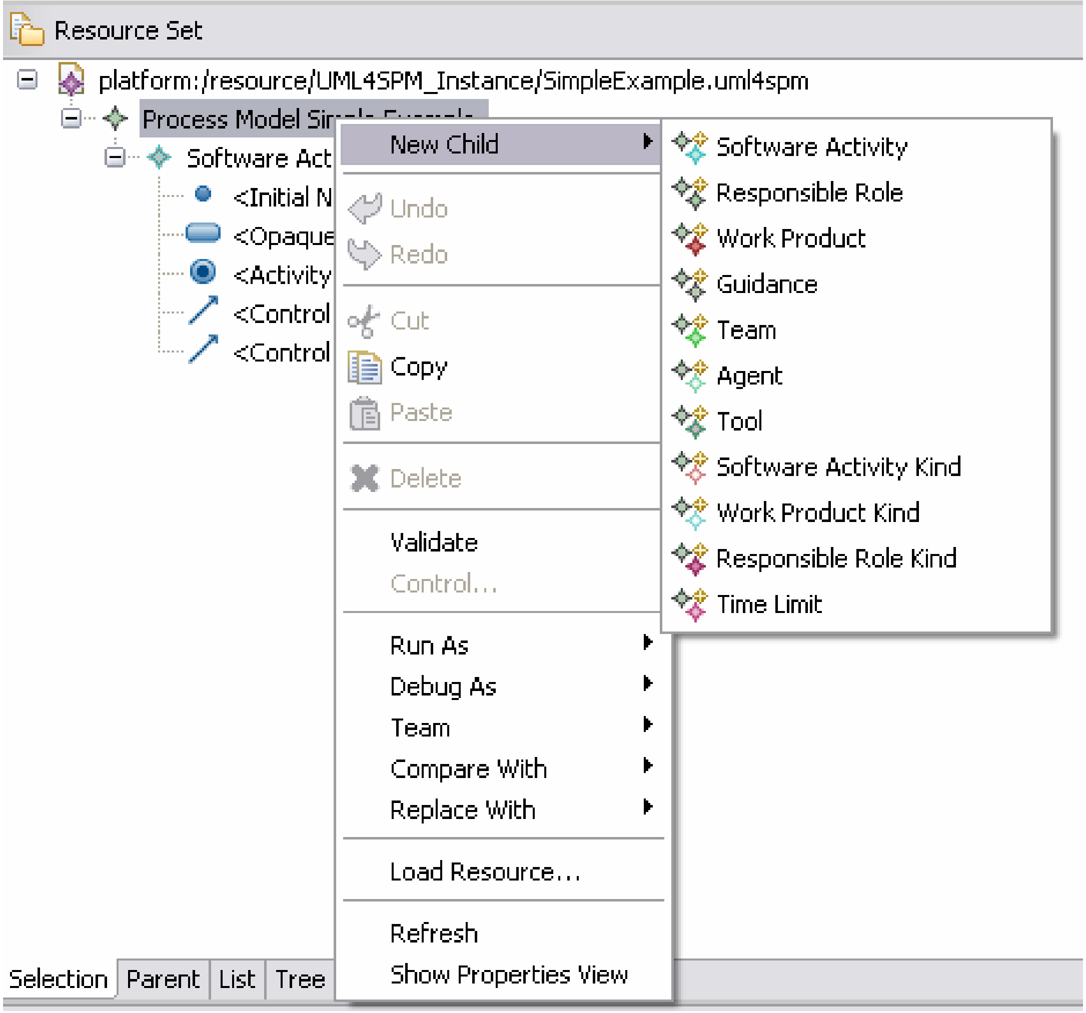


Figure 8.10. UML4SPM Process Model Editor

Once process models edited with the UML4SPM Process Model Editor, they can be directly executed using the UML4SPM Process Execution Engine, which we present in the following subsection.

4.2. UML4SPM Process Execution Engine

The UML4SPM Process Execution Engine is based on the UML4SPM *Executable Model* we presented in the previous section. It implements each of the class's operations defined in the executable model. The implementation we propose is in Java.

The process execution engine takes as input an UML4SPM process model and executes it according to the execution behavior defined in the UML4SPM executable model. The path to the process model can be defined in a file or given in an interactive way by the agent.

When the process model is loaded in memory, the process engine creates for each element in the model, its corresponding execution class. The mappings between the

UML4SPM elements and their execution classes are defined in an external configuration file in form of a pair of: UML4SPM element name=> Execution Class name (e.g. SoftwareActivity => ActivityExecution, InitialNode => InitialNodeExecution, etc.). The main goal behind this configuration is to give more **flexibility** in case of extending the UML4SPM language. Indeed, for instance, if a new kind of action (metaclass) has to be added to the UML4SPM metamodel, then, developers have simply to:

- Define the execution behavior of the new action metaclass in an execution class;

- Make the newly defined action execution class inheriting the *ActionExecution* abstract class. Thus, the new action will inherit the general behavior of actions (and transitively *ActivityExecutionNode*) and then can receive token offers, fire its input and output pins, etc;

- Add the entry NewMetaClass name=> Execution Class name in the configuration file;

- Put the new action execution class in the same workspace (location) of the other execution classes;

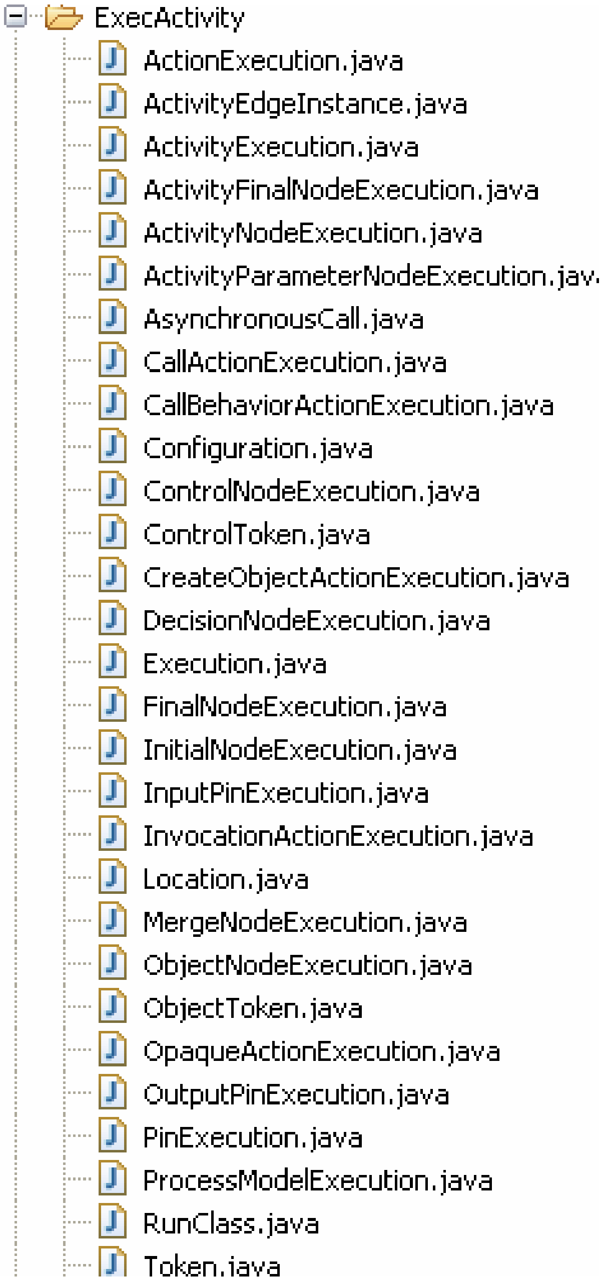
These steps apply to any new metaclass that have to be added to the UML4SPM metamodel whatever the metaclass it extends (*ActivityEdgeInstance*, *ControlNodeExecution*, *ObjectNodeExecution*, etc).

To start the execution of the process, the engine looks among the process's software activity executions, for the one with its *isInitial* property set to *true*. If no activity execution satisfies this requirement the process will not be executed. The same thing happens if more than one activity execution has its *isInitial* property set to *true*. If the process model contains only one software activity, than the process will be launched whatever the value of the software activity's *isInitial* property is.

When the process is launched, the execution of software activities and of their contents (i.e. *ActivityEdgeInstances*, *ControlNodeExecutions*, *ActionExecution* and *ObjectNodeExecutions*) follows the execution behavior described in the UML4SPM executable model. If a human interaction is needed, the Process Engine gives the hand to the Agent and wait for its entries in order to continue the execution. This case for instance with Decision Nodes without guards.

In the current implementation of the process engine we do not yet take in charge all the aspects related to resource management such as role affectations, WorkProduct versioning, etc.

Our process execution engine classes are given in figure 8.11. In the next section, we take the same example we used in the previous chapter and we execute it using the UML4SPM process execution engine.



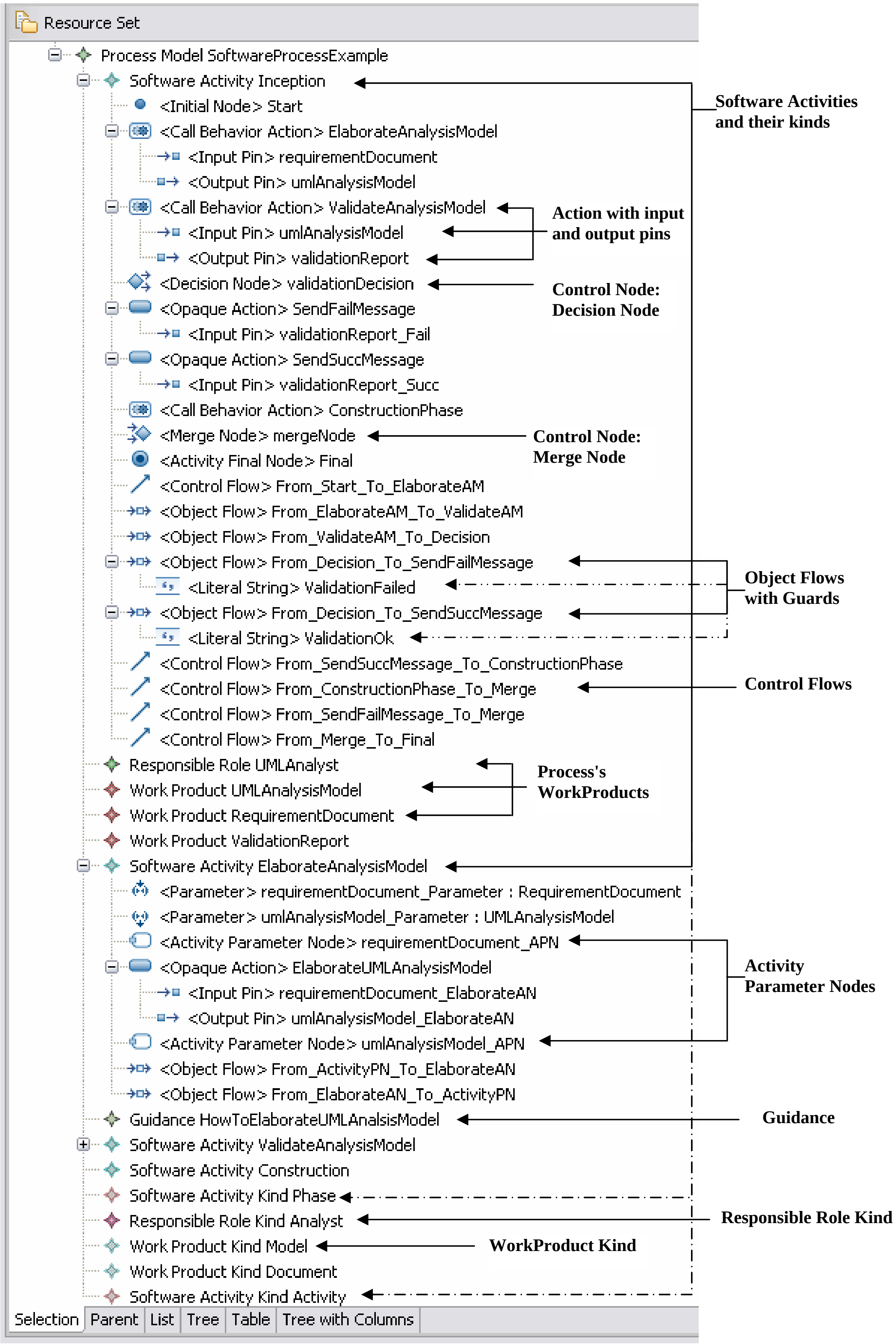
```
ExecActivity
    ActionExecution.java
    ActivityEdgeInstance.java
    ActivityExecution.java
    ActivityFinalNodeExecution.java
    ActivityNodeExecution.java
    ActivityParameterNodeExecution.jav
    AsynchronousCall.java
    CallActionExecution.java
    CallBehaviorActionExecution.java
    Configuration.java
    ControlNodeExecution.java
    ControlToken.java
    CreateObjectActionExecution.java
    DecisionNodeExecution.java
    Execution.java
    FinalNodeExecution.java
    InitialNodeExecution.java
    InputPinExecution.java
    InvocationActionExecution.java
    Location.java
    MergeNodeExecution.java
    ObjectNodeExecution.java
    ObjectToken.java
    OpaqueActionExecution.java
    OutputPinExecution.java
    PinExecution.java
    ProcessModelExecution.java
    RunClass.java
    Token.java
```

Figure 8.11. Process Execution Engine Classes

4.3. Software Process Example

In this section we show how the software process example we used in the previous chapter (UML4SPM-2-WSBPEL) approach is executed using the UML4SPM process engine. The process example modeled using the UML4SPM notations as well as its description are given in chapter 7, Section 6.2 of this document.

Figure 8.12 gives the whole process example modeled within the UML4SPM Process Editor. Some of the UML4SPM process model elements are highlighted on the figure.



```
Resource Set
  Process Model SoftwareProcessExample
    Software Activity Inception
      <Initial Node> Start
      <Call Behavior Action> ElaborateAnalysisModel
        <Input Pin> requirementDocument
        <Output Pin> umlAnalysisModel
      <Call Behavior Action> ValidateAnalysisModel
        <Input Pin> umlAnalysisModel
        <Output Pin> validationReport
      <Decision Node> validationDecision
      <Opaque Action> SendFailMessage
        <Input Pin> validationReport_Fail
      <Opaque Action> SendSuccMessage
        <Input Pin> validationReport_Succ
      <Call Behavior Action> ConstructionPhase
      <Merge Node> mergeNode
      <Activity Final Node> Final
      <Control Flow> From_Start_To_ElaborateAM
      <Object Flow> From_ElaborateAM_To_ValidateAM
      <Object Flow> From_ValidateAM_To_Decision
      <Object Flow> From_Decision_To_SendFailMessage
        <Literal String> ValidationFailed
      <Object Flow> From_Decision_To_SendSuccMessage
        <Literal String> ValidationOk
      <Control Flow> From_SendSuccMessage_To_ConstructionPhase
      <Control Flow> From_ConstructionPhase_To_Merge
      <Control Flow> From_SendFailMessage_To_Merge
      <Control Flow> From_Merge_To_Final
    Responsible Role UMLAnalyst
    Work Product UMLAnalysisModel
    Work Product RequirementDocument
    Work Product ValidationReport
    Software Activity ElaborateAnalysisModel
      <Parameter> requirementDocument_Parameter : RequirementDocument
      <Parameter> umlAnalysisModel_Parameter : UMLAnalysisModel
      <Activity Parameter Node> requirementDocument_APN
      <Opaque Action> ElaborateUMLAnalysisModel
        <Input Pin> requirementDocument_ElaborateAN
        <Output Pin> umlAnalysisModel_ElaborateAN
      <Activity Parameter Node> umlAnalysisModel_APN
      <Object Flow> From_ActivityPN_To_ElaborateAN
      <Object Flow> From_ElaborateAN_To_ActivityPN
    Guidance HowToElaborateUMLAnalsisModel
    Software Activity ValidateAnalysisModel
    Software Activity Construction
    Software Activity Kind Phase
    Responsible Role Kind Analyst
    Work Product Kind Model
    Work Product Kind Document
    Software Activity Kind Activity
```

Selection | Parent | List | Tree | Table | Tree with Columns

Software Activities and their kinds

Action with input and output pins

Control Node: Decision Node

Control Node: Merge Node

Object Flows with Guards

Control Flows

Process's WorkProducts

Activity Parameter Nodes

Guidance

Responsible Role Kind

WorkProduct Kind

Figure 8.12. Software Process Example modeled with UML4SPM Process Editor

We can notice for instance that software activities have a kind. The *Inception* software activity is of kind *Phase* while the *ElaborateUMLAnalysisModel* is of kind *Activity*. Similarly, WorkProducts and Responsible Roles have kinds. In the process example, we have the WorkProduct kind *Model* and *Document*, the Responsible Role *Analyst*, etc. Of course these kinds of process elements are domain specific and the process modeler can add as Software Activities, WorkProducts and responsible role kinds as necessary.

The second aspect we can highlight is the use of *CallBehaviorActions* to call other Software Activities from the *Inception Phase*. *CallBehaviorActions* have *Input* and *Output Pins* which are typed by the process's WorkProducts. *Object Flows* are used to connect between action's output pins and another action's input pins. In case of the of *Decision_To_SendFailMessage* and *Decision_To_SendSuccMessage* object flows, edges have guards. These guards will be evaluated at runtime by the process engine and the flow of work will depend on the result of guard evaluations.

Finally, in the case of the *ElaborateUMLAnalysisModel* and *ValidateAnalysisModel* software activities, *ActivityParameterNodes* are used to bring the data flow to the activity. These activities are called synchronously from the *Inception Phase*.

Software Process Model Execution

This software process example is passed to the UML4SPM process engine which executes it directly, without any intermediate step or configuration phase. The execution traces are given in the Appendix C of this document.

In the next section, we discuss the approach as well as the results we had using the UML4SPM Execution Model.

5. Discussion

The previous sections aimed at presenting the UML4SPM executable model rationale as well as the details of each of its classes and their implementations. In this section, we summarize all the aspects we addressed while discussing the important points of this approach.

- **UML4SPM Executable Model**: the UML4SPM Executable Model we propose is based on the *Executable UML Foundation* specification [OMG 06e], a work on progress at the OMG. During the implementation of this executable model in the context of UML4SPM, we realized that many aspects were lacking or not taken into account by *Executable UML*. We highlighted them and we proposed new operations on already existing execution classes and we also defined new execution classes in order to overcome these lacks. Of course, in addition to execution classes proper to UML4SPM, the operations and execution classes we newly defined concern UML2.0 *Activity* and *Action* package metaclasses and respect the execution semantics given by the UML2.0 standard;

- **UML4SPM Executable Model Implementation**: in addition to the proposition of an executable model in form of classes and operations, we give a Java implementation of this model. This implementation is then used as basis of the UML4SPM Process Execution Engine;

- **Reusability of the UML4SPM Execution Model for UML2.0 Activity Diagram Executions**: since UML4SPM is based on UML2.0 *Activity* and *Action* packages, the execution classes proper to these package elements we defined as well as their implementations are directly reusable for UML2.0 activity diagram executions;

- **Extensibility of the UML4SPM Language**: if process modelers need to extend the UML4SPM metamodel by adding for instance, a new kind of action, control node or object node, this can be done at lower costs. They simply have to define the execution class of the newly defined metaclass and make it extending the UML4SPM Execution Model class it tends to specialize. We gave all the details on UML4SPM executable model extension on section 4.2.

- **Strong Coupling of UML4SPM Process Models and their Execution Instances**: in the design of the UML4SPM Executable Model we put as a crucial requirement, the keeping of a strong coupling between process model elements and their execution instances. Thus, in the execution classes, we define only the behavior of UML4SPM elements. At runtime, when the execution class instance is created, it only keeps a reference to the process model element it defines its execution behavior. When the execution class instance requires a data, it takes it directly from the process model element definition. Thus, if the process model element evolves or has some of its element properties to be modified, the execution class instance will always has access to the correct (last) version of data. This facility opens some large perspectives such as the possibility to modify process models at runtime without restarting the execution of the process. Of course, the process model modification has to be performed from the API classes generated from the UML4SPM metamodel and under some conditions that have to be defined. The definition of these conditions is underway and goes beyond the scope of this document.

- **Connection with External Applications**: using the *OpaqueActionExecution* class we defined, it is rendered possible to execute a Java code expressed in the *body's* property of the *Opaque Action* without restarting the process. Details on how this is performed are given in section 3.7. Thanks to this facility, we can now envisage the possibility that the process model execution can be connected with an external application without interrupting the execution of the process. This can be for instance a simple Graphic User Interface that takes into account human interactions, a company's business applications and Workflows or a web-service based applications. We are currently exploring the feasibility and limits of these possibilities;

- **Concurrent Software Activities Executions**: the UML2.0 *Activity* and *Action* execution semantics supposes that some elements execute concurrently. In the context of UML4SPM process models, we can have the situation when two or more software activities or actions have to be launched simultaneously. This is the case for instance when a fork node is used to parallelize the execution of two or more workflows. Also, when a software activity calls another one asynchronously, the called activity has to start its execution while the calling one has to continue its own execution, both concurrently. In the UML4SPM Execution Model implementation, we take into account these constraints and we use Java Threads in order to ensure concurrency while executing UML4SPM process models.

- **Process Model Executions Through Interpretation**: UML4SPM process models are interpreted. Thanks to the strong coupling between UML4SPM process models and their execution behaviors, the execution is done in one pass. If the process model has to be modified during the process execution, there is no need to re-instantiate execution classes since they only represent the behavior of the process model element not its data, which is extracted directly from the element.

Another way to use the UML4SPM Executable Model is to define its execution behavior using the *Kermeta* executable meta-language instead of Java. *Kermeta* is developed at the IRISA laboratory and can be used to define the execution behavior at the metamodel level [Muller 05]. *Kermeta* extends EMOF (part of the MOF 2.0 specification) [OMG 06c], and allows that metaclasses be enriched with operations as well as their implementations. The investigation of the *Kermeta* initiative and its application to the UML4SPM Executable Model is underway. Our choice for Java for implementing the UML4SPM Executable Model was guided by efficiency reasons and by the possibility to reuse an already existing and powerful tooling support such as Eclipse/EMF development environment. Now that the *Kermeta* environment is mature and provides a large panoply of functionalities and analysis facilities, we envisage to implement the UML4SPM Execution Model using this executable meta-language.

Another way to use the UML4SPM Executable Model is to define a new package at the metamodel level called UML4SPM *Runtime Behavior*. This package will then merge the UML4SPM metamodel we already presented in chapter 5 of this document. The executable model will be reported on each of the *Runtime Behavior* metamodel metaclasses. The only condition is to have the Java classes representing (generated from) the metamodel. This can be obtained easily using the Eclipse development environment coupled with the EMF framework. Once the execution behavior (operations) implemented within the generated Java classes, developers have to make sure that these operations will not be overridden in case of a new code generation due to a modification of the metamodel. However this approach remains very ad-hoc and requires the manipulation of the Java code, to make sure that operations will not be replaced, etc. It also dependents too much on the development environment.

When writing this document, the UML4SPM Execution Model implementation still lacks support for event actions (i.e. *AcceptEventAction* and *SendEventAction*) as well as for the *Fork* and *Join* nodes. The definition of their execution classes and implementations is underway.

6. Conclusion

In this chapter we presented the UML4SPM *Execution Model* approach for process model executions. This approach aims at overcoming the limits of the UML4SPM-2-WS-BPEL that we introduced in the previous chapter.

As we saw, this approach offers some good perspectives. The first one is that process modelers have to deal with only one language for process modeling and execution. UML4SPM process models can be executed directly. Neither a model transformation nor intermediate steps are required. Another important point is that process models are strongly linked with their execution. Thus, process models can be modified at runtime without a need to restart or to interrupt the execution of the process. Also, we offer the possibility to UML4SPM process models to be linked with external applications which opens the way to many interesting possibilities.

In addition to the *Executable Model*, we proposed a Java implementation of this model. Then, we used it as a basis of the UML4SPM Process Execution Engine. We also define a UML4SPM Process Model Editor. Process models edited within this editor are straightforward executed by our process engine.

To demonstrate the feasibility of the approach, we used a complete software process example that we edited within the UML4SPM Process Model Editor and then we executed successfully with the UML4SPM Process Execution Engine. The process covered some typical software process characteristics such as automatic actions, decision points, synchronous and asynchronous activity calls, concurrency, interactions, etc.

Finally, we discussed in details the important aspects of this approach and we highlighted the remaining executable classes to realize.

A major perspective of this work is the integration of OCL as a language for expression Guards on activity edges in process models. This would add more expressiveness and powerfulness to process models. Another important aspect is resource management facilities which have to be integrated to the UML4SPM Process Engine. This would allow role affectations, workproducts versioning, etc. Finally, a more precise work has to be done in order to define upon which conditions process models can be modified during the execution of the process without interrupting its execution.

Chapter 9

Conclusions

In this thesis, we have addressed the problem of satisfying the apparently conflicting requirements of *Abstraction* and *Executability* in *software process modeling languages*. At this aim, we proposed UML4SPM, a UML2.0-Based Language for Software Process Modeling. Main contributions that led to this proposition can be summarized as follows:

- **Reusing the current state of the art**. While designing UML4SPM, we put a high interest in reusing and leveraging the lessons learned from the software process modeling community. Thus, our first step was to specify major requirements to satisfy when building a software process modeling language. These requirements were identified from well-known and approved works done in the literature. *Semantic richness*, *understandability*, *precision*, *modularization* and *executability* were retained as principal requirements. These requirements have been taken into account in the comparison we provide of current SPMLs and in defining our language, UML4SPM.

  One lesson we learned from first-generation SPMLs is that it is encouraged to use wide spread modeling languages with high level constructs as a basis of a SPML instead of proprietary and low level formalisms. This would ease much more its adoption. A high level of abstraction would help the very large number of process's actors with completely diverse backgrounds to reason and discuss the different aspects of the process. While the popularity of the language would avoid that people have to learn a new language and allows reusing the set of tools and training supports already provided. In this thesis we considered this recommendation and we investigated the reuse of UML, a standard and wide spread modeling language providing high level constructs, as a basis of our SPML [Bendraou 05a].

- **Definition of a framework for classifying and comparing between the different process technologies domains**. In front of the proliferation of process technology domains such as *BPM*, *SPE*, and *WfM*, we find it interesting to determine the frontiers between these different communities. Indeed, whether each community is evolving its process technology individually with its set of concepts and expectations for process modeling and execution, few works have been done in order to find commonalities/distinctions between these different research areas. In this thesis, we defined a framework that classifies and compares the different process technologies [Bendraou 07b]. This framework gives the process definition, characteristics, modeling objectives, process model constituents, process context and scope of each domain. It also clarifies the relationship between each of these domains.

- **Definition of an executable software process modeling language providing high level abstractions**. In this thesis we proposed UML4SPM, a UML2.0-based language for software process modeling and execution [Bendraou 05a]. UML4SPM comes in form of a MOF-Compliant metamodel, which extends the UML2.0 Superstructure standard [OMG 07b], a simple yet expressive graphical notation, and high level constructs with precise execution semantics.

For the definition of our SPML, we first identified an exact subset of UML2.0 concepts suitable for software process modeling. These concepts are principally parts of the *Activity* and *Action* packages and provide mechanisms for the sequencing of activities and actions, for expressing concurrency, conditions, iterations, events, exception handling, call actions, etc. We extend this subset of UML2.0 with the UML4SPM metamodel, which comes with a set of metaclasses with semantics proper to software process modeling (e.g., *Responsible Role, WorkProduct, Guidance, TimeLimit, Team, Tool, Agent*, etc.).

We also provide a notation for UML4SPM. That latter is inspired from UML2.0 *Activity* diagram notations. Some modifications were introduced in order to take into account some features proper to software process modeling but also to increase understandability. For UML2.0 *Activity* elements and *Actions* for which no notation is proposed by the standard, we proposed one.

Regarding ***executability*** of UML4SPM software process models, we investigated two approaches.

The UML4SPM-2-WSBPEL approach

For not starting from scratch, the first approach consisted in exploring the feasibility of using WS-BPEL as a target execution language for UML4SPM process models [Bendraou 07c]. WS-BPEL is a standard, plenty of tools are provided and is largely adopted by industrialists. Therefore, we introduced the language and its main features and we identified detailed mapping rules between UML4SPM concepts and WS-BPEL constructs. The mapping rules we proposed are not only UML4PSM-to-WS-BPEL specific since all rules that deal with UML2.0 concepts can be reused by any UML2.0-Based language or profile for process modeling. While identifying these mapping rules we have been confronted to some issues. Main ones are:

- UML4SPM elements with a semantic proper to software process modeling (i.e., *WorkProduct, Responsible Role, Guidance*, etc) have no equivalent in WS-BPEL;
- There is no one-to-one correspondence between elements of the two languages;
- BPEL's lack in supporting some control flow patterns (e.g., multiple merge, discriminator, etc.) and arbitrary cycles;

Another important point we addressed in this approach was the limit of WS-BPEL in supporting human interactions. We presented the different propositions for dealing with the human dimension such as the *Workflow Service* proposition. We also discussed the advantages and limits of each approach and we highlighted the urgent demand to standardize an interface of such *Workflow Service*.

For illustrating the approach, we gave a software process example that we modeled using UML4SPM. We then presented the main steps transforming the UML4SPM process model into WS-BPEL code.

As a general conclusion of this approach, we can enumerate its advantages and its limits. The first point is the undeniable advantage of to be able to reuse the myriad of WS-BPEL process engines and training supports provided by the Business Process Management community. The field is very mature and very active, which opens very large perspectives. We don't have to deal with all issues related to resource management, distribution, exceptions, etc. All these aspects are already incorporated within process engines. However, an important barrier of this approach is that process modelers have to deal with two languages, UML4SPM and BPEL. Additionally, for any changes in the process model, a new BPEL code generation has to be carried out and a configuration phase is required before deploying the process. Any modification in the BPEL code can't be traced-up to the UML4SPM process model which may lead to incoherencies between UML4SPM process models and the generated BPEL code.

In order to face these obstacles, we decided to explore another solution.

The UML4SPM Execution Model approach

The aim of this approach is to allow a straightforward execution of UML4SPM process models without any transformation or intermediate steps. The only condition is that process models are well formed. By well formed, we mean that process models should respect the structure and constraints defined in the metamodel.

To achieve this goal, in this thesis we proposed the UML4SPM *Execution Model*. This model defines execution behavior semantics of UML4SPM elements. Thus, for each UML4SPM metaclass having execution semantics, we defined its *execution class*, which at runtime, reproduces the execution behavior semantics of that metaclass. This execution semantics is expressed in terms of *operations* within the *execution classes*. We based our work on the *Executable UML Foundation*, a work on progress at the OMG [OMG 05c]. In this work we studied this specification and we drew from it the UML4SPM *Execution Model*. We also identified the set of operations and execution classes lacking by the *Executable UML Foundation* specification [OMG 06e]. Our *Execution Model* can be reused for executing UML2.0 *Activity* diagrams since UML4SPM extends UML2.0 *Activity* and *Action* concepts.

For the *Execution Model* we proposed, we provided a Java implementation for each of class of the model (the *Executable UML Foundation* does not provide any implementation). Classes of the model that represent execution semantics of UML2.0 elements have been implemented according to the semantics defined in the UML2.0 standard.

This implementation combined with the *Executable Model* principle offers some interesting facilities. The first one is that process modelers have to deal with only one language for process modeling and execution. UML4SPM process models can be executed directly without any configuration or refinement steps. A second important point is that process models are strongly linked with their execution. Thus, process models can be modified at runtime without a need to restart or to interrupt the execution of the process. Finally, we also offer the possibility to UML4SPM process models to be linked with external applications, which opens the way to many interesting possibilities.

One point that may penalize this approach is that we need to deal with all the aspects of resource management, role affectations, distribution, aspects that are already integrated with current BPM process engines. These aspects and the possibility to reuse some resources management facilities of current BPM process engines were not investigated yet in the context of this thesis.

- **Validation of the approach**. We evaluated UML4SPM with the set of SPMLs requirements defined by the literature. We saw that UML4SPM succeeded in fulfilling the majority of them. *Semantic richness* is provided thanks to a rich set of process elements we defined in the metamodel, to powerful mechanisms for activity and action coordination and sequencing borrowed from UML2.0, etc. *Modularization* is addressed by using the *CallBehaviorAction* as means to compose, to call or to coordinate between activity executions. The *Precision* requirement is reached thanks to the set of concepts such as *Software Activity, Action* and *Software Activity Kind* elements which allow the modeling of any process hierarchy. Regarding *Understandability,* undeniably, UML4SPM has a serious advantage since it reuses UML2.0 notation and diagrams. UML2.0 is wide-spread and many people are already familiar with its use.

  Another evaluation regarding the expressiveness of UML4SPM consisted in representing the well-known ISPW-6 software process example using our language. The process example comes in form of *core* problem and *optional* extensions. UML4SPM succeeded in modeling all process's activity aspects and issues related to the core problem. Additionally, we addressed main parts of optional extensions that relate the process executions. The evaluation of UML4SPM was validated in [Bendraou 06]

  For the execution of UML4SPM process models, in this work we provided a UML4SPM *Process Execution Engine*. That latter takes as input a UML4SPM process model edited with the UML4SPM *Process Model Editor* and directly executes it. Our process engine is based on the UML4SPM *Execution Model* we defined. For validating this process engine (transitively, the UML4SPM execution model), we tested it with a complete software process example that we edited within UML4SPM *Process Model Editor.*

Perspectives

The perspectives of this thesis can be separated into short-term and long-term perspectives.

Short-Term Perspectives

- **Regarding the UML4SPM-2-WSBPEL approach**, an important perspective in order to deal with human interactions would be the proposition of a standard interface definition of what we called *"Workflow Service".* Currently, services provided by tool vendors to deal with human interactions are proprietary and vendor's specific. A standard service interface would allow the homogenization of process executions across the different WSBPEL process engines.
- **Regarding the UML4SPM Execution Model approach**, in the current implementation we provide, guards on activity edges are expressed in a proprietary way (c.f., Chapter 8, Section 3.4) , and expressions can only be applied on *WorkProduct* and *Responsible Role* properties. In the near future, we plan to use OCL as a language for formulating such expressions. OCL is standard and offers powerful mechanisms for navigating models and for expressing constraints, conditions, invariants, etc. We will then integrate an OCL checker to our UML4SPM process execution engine. This would add more expressiveness and precision to UML4SPM process models.

In the near future, we plan to implement the UML4SPM Execution Model using the *Kermeta* executable meta-language which would allow UML4SPM process models not only to be executed but also to be analyzed thanks to the facilities offered by *Kermeta*.

We also envisage investigating the possibility of reusing resource management facilities offered by some process engines in the field of business process management. This would allow managing role affectations, workproducts versioning, alarms, etc.

Long-Term Perspectives

- **Process Model modification at runtime**. Using the UML4SPM Execution model approach and the implementation we provide, we saw that technically, it is rendered possible to modify the UML4SPM process model at runtime without affecting the process execution. Of course, to this end, some conditions have to be satisfied (e.g., the activity's constituent we need to modify must not have its owning activity's state at running or terminated or aborted, etc.). One major perspective would be to identify these conditions in a precise and formal way. This would allow an efficient use of process models and a means to make them evolve dynamically during the execution.
- **Interaction of process model executions with external applications**. In the UML4SPM Execution Model, we proposed the *OpaqueActionExecution* class, which allows the execution of Java code at runtime without interrupting or restarting the process execution. The behavior of the *OpaqueActionExecution* could be then used /specialized in order to define more specific actions. When writing this document, we identified three kinds of actions. The first one aims at specifying a standard GUI to be used in actions requiring human interactions. The second kind of action is to allow tool modeling service calls while using the ModelBus approach [Blanc 04], a work done in our team at LIP6 in order to allow interoperability between modeling tools. Finally, an action execution that allows calling distant web services from the process execution.
- **Fragmentation of process models**. Due to software system complexity, it is difficult to manage the development and maintenance of an entire system with a single software process. We consider handling this complexity with the notion of "software process partitioning". This notion consists in decomposing a complex software process into sub-processes and in allowing those sub-processes to be realized independently with different development sub-teams. To achieve this objective, we need to deal with the following difficulties: 1) managing the interactions between the sub software processes, 2) partitioning the entire software specification into different parts involved in each sub process, and 3) managing the concurrent work realized in each sub process. As a solution, we aim to base our work on top of ModelBus [Sriplakich 07a, 07b]. In fact, ModelBus offers functionalities for partitioning a large system specification, for supporting concurrent modifications in those partitions (with approach diff/merge), and for maintaining links between the partitions. We aim to use ModelBus to manage model partitions involved in each sub process, and support concurrent work realized by each sub process on models, and to manage the relationships between these sub-processes.

References

[ActiveBPEL] ActiveBPEL at http://www.active-endpoints.com/active-bpel-engine-overview.htm, last time page visit: February 2007

[Adams 03] M. Adams, D. Edmond and A ter Hofstede, “The Application of Activity Theory to Dynamic Workflow Adaptation Issues”, 7th Pacific Asia Conference on Information Systems, Adelaide, South Australia, July 2003

[Ambriola 94] Ambriola V., Conradi R. and Fuggetta A. “Experiences and Issues in Building and Using Process centered Software Engineering Environments”, Internal draft paper, Politecnico di Milano, September 1994.

[Ambriola 97] V. Ambriola, R. Conradi, and A. Fuggetta, “Assessing Process-Centered Environments”. ACM Transactions on Software Engineering and Methodology, 6(1), July 1997

[ANSI/IEEE 87] ANSI/IEEE Std 1012-1986, "IEEE Standard for Software Verification and Validation Plans", The Institute of Electrical and Electronics Engineers, Inc., February 10, 1987

[ApacheAgila] Apache Agila at http://wiki.apache.org/agila/, last time page visit: February 2007

[Armenise 93] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. A survey and assessment of software process representation formalisms. International Journal of Software Engineering and Knowllegde Engineering, 3(3):401–426, Sept. 1993

[Arlow 97] Arlow J., Bandinelli S., Emmerich W., and Lavazza L., "Fine Grained Process Modeling: an Experiment at British Airways, Software Process Improvement and Practice, J. Wiley, 3,2, 1997.

[AS CNRS 04] Action Scientifique CNRS, Rapport de Synthèse sur le MDA (Model-Driven Architecture), by J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.M. Favre, S. Gérard and J.M. Jézéquel, 2004

[Atkinson 03] Atkinson C., Kühne T., "Model-Driven Development: A Metamodeling Foundation", IEEE Software, September 2003

[Bandinelli 93] Bandinelli S., Fuggetta A. and Ghezzi C. "Software process model evolution in the SPADE environment". IEEE Transaction in Software. Engeneering. 19, 12 Dec. 1993, .1128–1144

[Bandinelli 95] Bandinelli S., Fuggetta A., Lavazza L., Loi M., and Picco G.P., Modeling and Improving an Industrial Sofiware Process, IEEE Transaction in Software. Engeneering, 21, May 1995.

[Bandinelli 96] Bandinelli S., Di Nitto E., and Fuggetta A., Supporting cooperation in the SPADE-1 environment, IEEE Transaction in Software. Engeneering, 22, 12, 1996.

[Bastida 05] L.Bastida Merino and G. Benguria Elguezabal1, "Business Process Definition Languages Versus Traditional Methods Towards Interoperability", Book chapter, COTS-Based Software Systems, Lecture Notes in Computer ScienceVolume 3412/2005, pages 25-35, ISBN:978-3-540-24548-3, January 2005

[Bastos 02] R. Bastos and D. Ruiz. Extending UML activity diagram for workflow modeling in production systems. In R. H. Sprague, Jr., editor, Proc. 35th Annual Hawaii Intern. Conference on System Sciences (HICSS-35). IEEE Computer Society, 2002.

[Becker 02] Becker, J., zur Muehlen, M. and Gille, M. (2002) ‘Workflow application architectures:classification and characteristics of workflow-based information systems’, in L. Fischer (Ed.) Workflow Handbook 2002. Future Strategies, Lighthouse Point, FL, pp.39–50

[Bendraou 05a] Bendraou R., Gervais M.P. and Blanc X., "UML4SPM: A UML2.0-Based metamodel for Software Process Modeling", in Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05), Montego Bay, Jamaica, Oct. 2005, LNCS, Vol. 3713, PP 17-38.

[Bendraou 05b] Bendraou R., Desfray P., and Gervais M.P., "MDA Components: A Flexible Way for Implementing the MDA Approach", in Proceedings of the European Conference on Model Driven Architecture –Foundations and Applications (ECMDA-FA'05), Nuremberg, Germany, Nov. 2005, LNCS Vol. 3748, PP 59-73.

[Bendraou 06] Bendraou R., Gervais M.P. and Blanc X, "UML4SPM: An Executable Software Process Modelling Language Providing High-Level Abstractions", in Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), pp. 297-306, Hong Kong, China, , 2006

[Bendraou 07a] Bendraou R. , Desfray P., Gervais M.P., and Muller A., "MDA Tool Components: A Proposal for Packaging Know-how in Model Driven Development", to appear in SoSyM: Journal on Software & System Modeling. LNCS 2007

[Bendraou 07b] Bendraou R., Gervais M.P. "A Framework for Classifying and Comparing Process Technology Domains", to appear in Proceedings of International Conference on Software Engineering Advances (ICSEA'07). IEEE Computer Society Press 2007

[Bendraou 07c] Bendraou R. , Sadovykh A. Gervais M.P. and Blanc X,, " Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach ", to appear in Proceedings of the 33rd EUROMICRO Conference of Software Engineering Advanced Application (SEAA). IEEE Computer Society Press 2007.

[Bézivin 01] Bézivin, J., Gerbé, O. Towards a precise definition of the OMG/MDA framework. ASE'01, Automated Software Engineering, San Diego, USA, November 26-29, 2001.

[Bézivin 05] Bézivin, J., On the unification power of models, Software and Systems Modeling, Volume 4, Issue 2, May 2005, pages 171 – 188

[Blanc 04] Blanc X., Gervais M.P., and Sriplakich P. "Model Bus: Towards the Interoperability of Modelling Tools", in Proceedings of the Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping University, Sweden, June 2004

[Boehm 76] B. Boehm, "Software Engineering". In IEEE Transactions. Computer, C-25, 12, 1226-1241, 1976.

[Boehm 87] B. Boehm, "A Spiral Model of Software Development and Enhancement". Computer, 20(9), 61-72, 1987

[Bolcer 98] G. A. Bolcer and R. N. Taylor, "Advanced workflow management technologies," Software process - Improvement and practice, vol. 4, pp.125-171, 1998.

[Bordbar 04] Bordbar B., Staikopoulos A.: "On Behavioural Model Transformation in Web Services", Proceedings of the ER 2004 Workshops CoMoGIS, COMWIM, ECDM, CoMoA, DGOV, and ECOMO, Shanghai, China 2004, Springer Press, 2004.

[BPEL4WS 03] BEA, IBM, Microsoft, SAP and Siebel, “Business Process Execution Language for Web Services Version 1.1”, S. Thatte, et al., May 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[Brinkkemper 01] Brinkkemper, S., Saeki, M., and Harmsen. F. A Method Engineering Language for the Description of Systems Development Methods. 13th Conference on Advanced Information Systems Engineering (CaiSE’2001), Lecture Notes in Computer Science 2068, pp. 473–476, 2001

[Cass 00] Cass, A.G., Staudt Lerner, B., McCall, E.K., Osterweil, L. J.,Sutton, Jr., S.M., and Wise, A., "Little-JIL/Juliette: A Process Definition Language and Interpreter" .In Proceedings of the 22nd Internantioanl Conference on Software Engineering, June 2000.

[Cohen 88] Cohen D. "AP5 Manual". Univ. of Southern California, Information Sciences Institute, March 1988.

[Combemale 06] Combemale B., Caplain A., Crégut X. , and Coulette B., *"Towards a rigorous use of SPEM*" in Proceedings Of ICEIS'06, INSTICC, May, 2006 - Paphos, Cyprus

[Conradi 92a] Conradi, R., Fernström, C., Fuggeta, A., Snowdon,B., "Towards a Reference Framework for Process Concepts", Proc. Second European Workshop on Software Process Technology, Trondheim, 9/1992, Lecture Notes in Computer Science, 635, Springer-Verlag Ed.

[Conradi 92b] Conradi R. et al. "Design, use, and implementation of SPELL, a language for software process modeling and evolution". In Proc. of EWSPT'92, Springer Verlag LNCS vol.

[Conradi 94] R. Conradi, J. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, C. Liu, “EPOS: Object-Oriented and Cooperative Process Modelling”. In A.Finkelstein, J. Kramer, and B. Nuseibeh, editors. Software Process Modelling and Technology. Research Studies Press Limited (J. Wiley), 1994

[Conradi 95] R. Conradi and C. Liu, “Process Modelling Languages: One or Many?”, in W. Schafer, ed., Proceedings of the 4th European Workshop on Software Process Technology (EWSPT-4), Noordwijkerhout, The Netherlands. Lecture Notes in Computer Science, Vol. 913, Springer, April 1995.

[Conradi 99] R. Conradi, M.J. Jaccheri, "Process Modelling Languages". In: Derniame, J.C., Kaba, B.A., Wastell, D. (eds.): Software Process: Principles, Methodology and Technology. Lecture Notes in Computer Science, Vol. 1500. Springer-Verlag,Berlin Heidelberg New York (1999) 27-51

[Chou 00] Chou, S.C. and Chen, J.Y.J., "Process Program Development Based on UML and Action Cases, Part 1: the Model", in Journal of Object-Oriented Programming, Vol. 13, Num. 2, pp 21--27, 2000.

[Chou 02] S.-C. Chou, A process modeling language consisting of high level UML diagrams and low level process language, Journal of Object Technology 1, 2002, 4, pp. 137–163

[Clarck 02] Clark T., Evans A., and Kent S. "A metamodel for package extension with renaming". In Jean-Marc Jézéquel, Heinrich Hussmann,and Stephen Cook, editors, 5th International conference on the Unified Modelling Language (UML 2002), volume 2460. Lecture notes in computer science, 2002.

[Cugola 98a] Cugola G. and Ghezzi C. , “Software processes: a retrospective and a path to the future,” Software process - Improvement and practice, vol. 4, pp.101-123, 1998.

[Cugola 98b] Cugola G., "Inconsistencies and deviations in process support systems". PhD Thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, February 1998

[Cugola 01] CUGOLA, G., DI NITTO, E., AND FUGGETTA, A., "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS", IEEE Transactions on Software Engineering. 2001

[Curtis 92] W. Curtis, M. I. Kellner and J. Over. "Process Modelling". Communication of ACM, 35 (9), 1992, pp. 75-90.

[Dami 98] Dami S., Estublier J., and Amiour M., “APEL: a graphical yet executable formalism for process modeling”, Automated Software Engineering Journal, special issue on Process Technology, vol. 5, no. 1, January 1998.

[Davenport 93] T. Davenport, Process Innovation: Reengineering work through information technology, Harvard Business School Press, 1993, Boston

[Desfray 99] Desfray P. and al, "White Paper on the Profile mechanism", OMG document ad/99-04-07,Avril 1999.

[Desfray 00 Desfray P., "UML profiles versus metamodel extensions: an ongoing debate". Available at http://www.omg.org/news/meetings/workshops/presentations/uml_presentations/5-3%20Desfray%20-%20UMLWorkshop.pdf

[Di Nitto 02] Di Nitto E. et at. "Deriving executable process descriptions from UML", in Proceedings of the 24th Inter. Conf. on Software Engineering (ICSE'02), Orlando, Florida 2002, ACM Press

[Dobson 06] Dobson G., "Using WS-BPEL to Implement Software Fault Tolerance for Web Services", in Proceedings of the 32nd EUROMICRO-SEAA'06 conference, IEEE Computer Society, 2006.

[Dowson 91] M. Dowson, B. Nejmeh, W. Riddle, "Fundamental Software Process Concepts", Proc. First European Workshop on Software Process Modeling, Milan, April 91, AICA Press.

[D'Souza 99] D'Souza D.F and Wills A.C. "Objects, Components, and Frameworks with UML: the Catalysis Approach". Addison-Wesley, 1999.

[Emerson 04] Emerson M., Sztipanovits J., and Bapty T., "A MOF-Based Meta-modeling Environment," Journal of Universal Computer Science, October 2004, pp. 1357--1382.

[EMF] Eclipse EMF (Eclipse Modeling Framework), at http://www.eclipse.org/emf/

[Eclipse] Eclipse Projects at http://www.eclipse.org/

[Endl 98] Endl, R., Knolmayer, G. and Pfahrer, M. (1998), 'Modeling Processes and Workflows by Business Rules' in 1st European Workshop on Workflow and Process Management (WPM'98), Swiss Federal Institute of Technology (ETH), Zurich

[EPF] Eclipse Process Framework (EPF), at www.eclipse.org/epf/

[Feiler 93] Feiler P.H., Humphrey Watts. S. "Software process development and enactment", in Proc. of 2nd Inter. Conf. on the Software Process, Berlin, 1993, IEEE Computer Society Press.

[France 06] France, R.B., Gosh, S. Dinh-Trong, T, and Solberg, A. "Model-driven development using UML2.0: Promises and Pitfalls, IEEE Computer Magazine February 2006, pp 59-66

[Franch 97] Franch, X.; Botella, P.; Burgués, X.; Ribó, J.M.: "ComProLab: A Component Programming Laboratory". Proceedings 9th Software Engineering and Knowledge Engineering Conference (SEKE), Knowledge Systems Institute, Skokie (1997), 397-406

[Franch 98] Franch, X.; Ribó, J.M., "A Structured Approach to Software Process Modelling", Proceedings 24th EUROMICRO Conference, IEEE Computer Society Press, Los Alamitos Washington Brussels Tokyo (1998), 753-762

[Fuentes 04] Fuentes L. and Vallecillo A.. "An introduction to UML profiles". UPGRADE, The European Journal for the Informatics Professional, 5(2):5–13, Apr. 2004

[Fuggetta 00] A. Fuggetta. "Software Process: A Roadmap". 22nd International Conference on Software Engineering (ICSE'2000), Future of Software Engineering Track, June 4–11, Limerick (Irlanda), ACM, 2000

[Gamma 94] Gamma E., Helm R., Johnson R., and Vlissides J. Design Patterns: Elements of Reusable Object- Oriented Software. Addison-Wesley, 1994.

[Georgakopoulos 95] Georgakopoulos D., Hornick M., Sheth A. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure", Distributed and Parallel Databases, vol.3, pp.119-152, 1995

[GEF] Eclipse GEF (Graphical Editing framework), at http://www.eclipse.org/gef/

[GMT] Eclipse GMT (Generative Modeling technologies), at http://www.eclipse.org/gmt/

[Giaglis 01] Giaglis, G.M. 2001. "A taxonomy of business process modeling and information systems modeling techniques". International Journal of Flexible Manufacturing Systems, Vol. 13, No. 2, 209-228.

[Gordijn 00] Gordijn J., Akkermans J. M. & Vliet J. C."Business Modeling is not Process Modeling", eCOM2000 workshop, 19 th International Conference on Conceptual Modeling 2000

[Greenfield 03] Greenfield J., Short K. "Software factories: assembling applications with patterns, models, frameworks and tools", in Proceedings of the 18th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Anheim, CA, USA, 2003, ACM press

[Gruhn 92] Gruhn, V. “Software Processes are Social Processes,” Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Quebec, Canada, 1992, pp. 196-201.

[Hammer 96] Hammer, M.: Beyond Reengineering – How the process-centered organization is changing our work and our lives. Harper Collins Publishers, 1996.

[Harel 00] D. Harel, “From play-in scenarios to code: An achievable dream,” Computer, to appear. Preliminary version in Tom Maibaum (Ed.), Proc. Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science, Vol. 1783, Springer-Verlag, 2000, pp. 22–34, IEE Computer 34:1, Jan. 2001, pp. 53–60.

[Hausmann 05] Hausmann J.H., Störrle H., "Towards a Formal Semantics of UML 2.0 Activities", in Proc. of the German Software Engineering Conference (SE'05).

[Henderson 94] Henderson, P. “Software Processes are Business Processes Too,” Proceedings of the Third International Conference on the Software Processes: Applying the Software Process, Reston, Virginia, 1994, pp. 181-182.

[Henderson 04] B Henderson-Sellers, CA González-Pérez, "A Comparison of Four Process Metamodels and the Creation of a New Generic Standard", Information and Software Technology, 2004

[Hosier 61] W. A. Hosier, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming", IRE Trans. Engineering Management, EM-8, June 1961

[Humphrey 89a] W.S. Humphrey. "The Software Engineering Process: Definition and Scope", in Proceedings of the 4th International Software Process Workshop on Representing and Enacting the Software Process, Devon, United Kingdom, 1989

[Humphrey 89b] W.S. Humphrey and M.I. Kellner, "Software process modeling: Principles of entity process models". In Proceedings of the Eleventh International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 1989, pp. 331-342.

[Humphrey 92] W.S. Humphrey and P.H. Feiler, "Software process development and enactment: Concepts and definitions". Tech. Rep. SEI-92-TR-4. Pittsburgh: Software Engineering Institute, Carnegie Mellon University. To be published, 1992.

[Hyungwon 96] Hyungwon L. and Chisu W. "HI-PLAN: A Structured Project Planning Method". In Journal of Korea Information Science Society, Vol. 23, No. 8, 821-831, Aug. 1996.

[IEEE 90] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990

[IBM 97] IBM Object Technology Center, "Developing Object-oriented Software: An Experience-based Approach" , Prentice-Hall, 1997, pp. 192-232.

[IBM 04] Summit Ascendant, at http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/may04/TheRationalEdge_May2004.pdf

[Intalio] Intalio BPMS Community Edition, at www.intalio.com

[ISO 06] ISO SEMDM, "Software Engineering — Metamodel for Development Methodologies", ISO document, ISO/JTC 1/SC 7 ICS 35.080, May 2006

[ISO 98] International Organization for Standardization. ISO/IEC, ISO 9000 Quality Management, 7th edition, 1998

[Jaccheri 99] Jaccheri M.L., Baldi M., Divitini M., "Evaluating the Requirements for Software Process Modelling Languages and Systems", in Proceedings of Process support for Distributed Team-based Software Development (PDTSD'99), Florida, USA, August 1999

[Jäger 98] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using UML for Software Process Modeling. Number 1687 in LNCS, pages 91-108, 1998.

[Jennings 96] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. "Agent-based business process management". International Journal of Cooperative Information Systems, p 105-130, 1996

[Johansson 93] H.J. Johansson, et.al, Business Process Reengineering: BreakPoint Strategies for Market Dominance, 1993, John Wiley & Sons

[Juric 07] Juric M., Todd D.H., "BPEL Processes and Human Workflow", SOA Web Services Journal,. 12-04-2006. http://webservices.sys-con.com/read/204417.htm last time page visite June 2007

[Kaiser 90] Kaiser G.E., Barghouti N.S. and Sokolsky M.H. "Preliminary experience with process modeling in the Marvel software development environment kernel". In Proceedings of the 23d Annual Hawaii Internernational. Conference on System Sciences, Vol.H Software Track. IEEE Computer Society, Washington, DC, 1990, 131-140

[Kellner 89] M. I. Kellner, "Software process modeling: Value and experience". SEI Tech. Rev. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1989, pp. 22-54.

[Kellner 91a] Kellner, M.I. and Rombach, H.D. "Session summary: Comparisons of software process descriptions". In Proceedings of the 6th International. Software Process Workshop. IEEE Computer Society, Washington, DC, 1991, pp. 7-18

[Kellner 91b] Kellner, M.I., Feiler, P.H., Finklestein, A., Katayama, T., Osterweil, L.J., Penedo, M.H., Rombach, H.D. "ISPW-6 software process example". In Proceedings of the first International. Conference on the Software Process. IEEE Computer Society, Washington, DC, 1991, pp. 176-186.

[Kloppmann 05] Kloppmann, M. et al. "WS-BPEL Extension for People BPEL4People", Joint white paper, IBM and SAP, July 2005.

[Korherr 06] Korherr B. and List B. "Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL". 2nd International Workshop on Best Practices of UML (BP-UML'06) at the 25th International Conference on Conceptual Modeling (ER'06), November 2006, Tucson.

[Kruchten 03] Kruchten, P, "The Rational Unified Process: An Introduction" Addison-Wesley Professional, 2003

[Lawrence 97] P. Lawrence (ed.), "WfMC Workflow Handbook", John Wiley & Sons Ltd., 1997.

[Lei 97] Lei, Y. and Singh, M.P.: A Comparison of Workflow Metamodels. Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling, Los Angeles, November 1997

[List 06] List B., Korherr B., "An Evaluation of Conceptual Business Process Modelling Languages", Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), April, Dijon, France, ACM Press, 2006.

[Lonchamp 93] Lonchamp, J. 1993. A structured conceptual and terminological framework for software process engineering. In Proceedings of the 2nd International Conference on the Software Process (ICSP 2) (Berlin, Germany). IEEE Computer Society Press, Los Alamitos, Calif.

[Loui 88] Loui, M.C., "The case for assembly language programming", in journal of IEEE Transactions on Education, Vol 31, 1988

[Madhavji 93] Madhavji N.H. and Penedo M.H editors. IEEE Transaction on Software Engineering: Special Issue on process evolution. IEEE Computer Society Press, December 1993

[Mantell 05] Mantell, K. "From UML to BPEL". URL: http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel, September 2005.

[Mellor 02] Mellor S.J. and Balcer M., "Executable UML: A Foundation for Model-Driven Architecture", Addison-Wesley, 2002

[Mellor 03] Mellor S.J. , A.N. Clark, and T. Futagami, “Model-Driven Development,” IEEE Software, no. 5, pp. 14-18, Sept./Oct. 2003.

[Modelware] Modelware, IST European Project contract no 511731, at http://www.modelware-ist.org/

[Modelplex] Modelplex, IST European Project contract IST-3408

[Montangero 99] Montangero C., Derniame J.C., and Kaba B.A., Warboys B. "The software process: Modelling and technology", LNCS GmbH. Vol. 1500/1999.

[MSProject] Microsoft Project Manager at http://www.microsoft.com/france/office/2007/solutions/epm/overview.mspx

[Mühlen 99] Mühlen. M. zur, "Evaluation of Workflow Management Systems Using Meta Models". In R.Sprague, Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS’99). 1999.

[Muller 05] Muller P.A,Fleurey F., and Jézéquel J.M. "Weaving executability into object-oriented meta-languages.", In S. Kent L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264--278, Montego Bay, Jamaica, October 2005. Springer.

[OASIS] http://www.oasis-open.org/

[Objecteering] Objecteering, at http://www.Objecteering.com

[Odell 94] Odell J. "Power Types". Journal of Object-Oriented Programming, 1994. 7(2): 8-12.

[OMG 00a] OMG, "Workflow Management Facility Specification v1.2", OMG document formal/00-05-02, April 2000, at http://www.omg.org.

[OMG 00b] OMG UML1.3, "Unified Modelling Language", version 1.3., OMG document formal/00-03-01, March 2000, at http://www.omg.org.

[OMG 01] OMG UML1.4, "Unified Modelling Language", version 1.4., OMG document formal/01-09-67, September 2001, at http://www.omg.org.

[OMG 02] OMG SPEM1.0, “Software Process Engineering Metamodel”, OMG document formal/02-11/14, November 2002, at http://www.omg.org.

[OMG 03] OMG MDA Guide Version 1.0.1., Object Management Group, June 2003. Document number omg/2003-06-01.

[OMG 04] OMG SPEM2.0 RFP, “Software Process Engineering Metamodel”, OMG document ad/2004-11-04, November 2004, at http://www.omg.org/docs/ad/04-11-04.pdf, page last visit January 2, 2007.

[OMG 05a] OMG SPEM1.1, “Software Process Engineering Metamodel”, OMG document formal/05-01-06, January 2005, at http://www.omg.org.

[OMG 05b] OMG XMI, "XML Metadata Interchange", version 2.1., OMG document formal/05-09-01 , September 2005 at http://www.omg.org

[OMG 05c] OMG, Semantics of a Foundational Subset for Executable UML Models RFP, OMG document ad/05-04-02, April 2005, at: http://www.omg.org/docs/ad/05-04-02.pdf, page last visit May 27, 2007

[OMG 06a] OMG BPMN, Business Process Modeling Notation final adopted specification, OMG document dtc/06-02-01, February 2006 at http://www.omg.org

[OMG 06b] OMG OCL, "Object Constraint Language version 2.0", adopted specification, OMG document formal/06-05-01, May 2006, at http://www.omg.org

[OMG 06c] OMG MOF, "Meta Object Facility version 2.0", adopted specification, OMG document formal/06-01-01, January 2006, at http://www.omg.org

[OMG 06d] OMG "Diagram Interchange", adopted specification, OMG document formal/06-04-04, April 2006, at http://www.omg.org

[OMG 06e] OMG, Semantics of a Foundational Subset for Executable UML Models, Initial Submission, OMG document ad/06-05-02 at: http://www.omg.org/docs/ad/05-04-02.pdf, page last visit July 1, 2007

[OMG 07a] OMG UML, "Unified Modeling Language", Infrastructure Specification, version 2.1.1., OMG document formal/07-02-06 , February 2007 at http://www.omg.org

[OMG 07b] OMG UML, "Unified Modeling Language", Superstructure Specification, version 2.1.1., OMG document formal/07-02-04 , February 2007 at http://www.omg.org

[OMG 07c] OMG SPEM2.0, “Software Process Engineering Metamodel”, OMG document, final adopted specification, ptc/07-03-03, March 2007, at http://www.omg.org.

[OMG UMLpf] OMG UML Profiles at http://www.omg.org/technology/documents/profile_catalog.htm

[Oracle] Oracle BPEL Process manager, at www.oracle.com/technology/bpel

[Osellus] Osellus IRIS Suite, at www.Osellus.com

[Osterweil 87] Osterweil L., "Software Processes Are Software Too" in Proc. of the 9th International Conference on Software Engineering (ICSE'9), New York, 1987, ACM Press.

[Ould 95] Ould, M.A., Business Processes: Modelling and analysis for re-engineering and improvement, John Wiley & Sons, Chichester, England, 1995.

[Ouyang 06] Ouyang, C., Dumas, M., Breutel, S., ter Hofstede, A.H.M.: "Translating Standard Process Models to BPEL". In Pohl, K., ed.: 18th Conference on Advanced Information Systems Engineering, Luxembourg, Springer (2006) forthcoming

[Paulk 95] M. Paulk, et al., "The Capability Maturity Model: Guidelines for Improving the Software Process", Addison-Wesley, Reading, MA, 1995, ISBN 0-201-54664-7

[Peltier 02] Peltier M., "Transformation entre un profile UML et un méta-modèle MOF: Application au langage MTRANS", LMO (Langages et Modèles à Objets), volume 1-n°1/2002, Hermes, 2002

[Perry 89] Perry D. E., Editor, Proc. of the 5th Inter. Software Process Workshop (ISPW’5), Kennebunkport, Maine, USA, October 1989, IEEE Computer Society Press

[Porter 85] Porter, M., 1985. Competitive Advantage, Fee Press, New York.

[Raistrick 04] Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I. "Model Driven Architecture with Executable UML", Cambridge University Press, 2004.

[RMC] IBM Rational Method Composer (RMC), at www.ibm.com/software/awdtools/rmc/

[RPM] Rational Portofolio Manager (RPM), at http://www-306.ibm.com/software/awdtools/portfolio/index.html

[RPW] Rational Process Workbench (RPW) at http://www-128.ibm.com/developerworks/rational/library/6001.html#author

[Riddle 89] W.E. Riddle, "Session summary: Opening session". In Proceedings of the 4th International Software Process Workshop. IEEE Computer Society, Washington, DC, (1989), pp. 5-10.

[Rolland 93] C. Rolland, Modeling the Requirements Engineering Process, 3rd European-Japanese Seminar on Information Modelling and Knowledge Bases, Budapest, Hungary, June 1993.

[Rolland 98] C. Rolland, A Comprehensive View of Process Engineering. Proceedings of the 10th International Conference on Aided Software Engineering CAiSE'98, B. Lecture Notes in Computer Science 1413, Pernici, C. Thanos (Eds), Springer. Pisa, Italy, June 1998

[Rothenberg 89] Rothenberg, J. "The Nature of Modeling in Artificial Intelligence, Simulation, and Modeling". In L.E. William, K.A. Loparo, N.R. Nelson, eds. New York, John Wiley and Sons, Inc., 1989, pp. 75-92

[Royce 70] W. W. Royce, "Managing the Development of Large Software Systems". In Proc. 9th. International Conference on Software Engineering, IEEE Computer Society, 1987 ,328-338 Originally published in Proc. WESCON, 1970.

[Ruiz 04] F. Ruiz-Gonzalez and G. Canfora "Software Process: Characteristics, Technology and Environments" UPGrade, The European Journal for the Informatics Professional, vol 5, no. 5, 2004, pp. 6-10

[Russel 06] Russell N., Van der Aalst W.M.P., Ter Hofstede A.H.M. , and Wohed P."On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling", In M. Stumptner, S. Hartmann, and Y. Kiyoki, editors, Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006), volume 53 of CRPIT, pages 95-104, Hobart, Australia, 2006. ACS

[Sarstedt 06] Sarstedt, S. "Semantics Foundation and Tool Support for Model-Driven Development with UML2 Activity Diagrams", PhD Dissertation, Ulm University, 2006

[Seidewitz 03] Seidewitz, E.: "What models mean". IEEE Software. 20(5), 26–32, (2003)

[Scheer 99] Scheer A.-W.: "ARIS - Business Process Modeling". 2nd ed. Berlin et al. (1999)

[Scacchi 01] W. Scacchi "Process Models in Software Engineering". In, J.J. Marciniak (ed.), Encyclopedia of Software Engineering, 2nd Edition, John Wiley and Sons, Inc, New York, Dec. 2001

[Scacchi 94] Scacchi, W. "Business Processes Can Be Software Too: Some Initial Lessons Learned," Proceedings of the Third International Conference on the Software Processes: Applying the Software Process, Reston, VA, 1994, pp. 183-184.

[Schmidt 98] M.-T. Schmidt, "Building Workflow Business Objects," in Business Object Design and Implementation II, D. Patel, J.Sutherland, and J. Miller, eds, Springer-Verlag, London,1998, pp. 64-76.

[Schreyjak 98] Schreyjak, S., "Synergies in a Coupled Workflow and Component-Oriented System". In: Grundy, John (Hrsg.): Proceedings of CBISE'98 CAiSE*98 Workshop on Component Based Information Systems Engineering, 1998.

[Sellic 03] Selic, B., "The pragmatics of model-driven development". IEEE Software 2003., 19–25, Special issue on model-driven development.

[Sommerville 07] I.Sommerville, "Software Engineering 8", 2007, eighth edition, Addison-Wesley, ISBN 10:0-321-31379-8

[Sol 92] Sol, H.G. and Crosslin, R.L. (Eds.) "Dynamic Modeling of Information Systems, II, North-Holland, Amsterdam, 1992

[Sriplakich 07a] Sriplakich P., X. Blanc, M.-P. Gervais, Support collaboratif pour la manipulation de modèles à large échelle, Actes des 3ème Journées sur l'Ingénierie Dirigée par les Modèles (IDM 07), 2007.

[Sriplakich 07b] Sriplakich P. , X. Blanc, M.-P. Gervais, ModelBus: a distributed platform for collaborative software engineering on large-scale models, Submitted to the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA), 2007

[Standish 06] Standish Group: "2006 Research Report" at: http://www.standishgroup.com.

[STL 06] STL: UML2.0 Semantics Project, at http://www.cs.queensu.ca/~stl/internal/uml2/index.html, page last visit: February 27, 2006

[Störrle 04] Störrle H. "Semantics of UML2.0 Activities with Data-Flow", in Proc. of the Visual Languages and Formal Methods Workshop (VLFM'04), Rome, Italy, Septembre 2004.

[Sutton 95a] Sutton, Jr., S.M., Tarr, P.L., and Osterweil, L.J., " An Analysis of Process Languages" CMPSCI Technical Report 95-78, University of Massachusetts, (1995)

[Sutton 95b] Sutton , Jr., S.M., Heimbigner D., and Osterweil L. J. "APPL/A: A language for software-process programming". ACM Transaction on Software Engineering and Methodology, 4(3):221–286, July 1995

[Sutton 97] Sutton, Jr. S. M. and Osterweil L. J. . The design of a next-generation process language. Technical Report CMPSCI Technical Report 96-30, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003, May 1996. Revised January, 1997.

[Swenson 95] Swenson K D and Irwin K 1995 Workflow technology:tradeoffs for business process re-engineering Conf. onOrganizational Computing Systems (Milpitas, CA, 1995) (New York: ACM) pp 22–9

[Sztipanovits 95] Sztipanovits, J., et al. "Multigraph : an architecture for model-integrated computing" .In ICECCS, pages 361–368, 1995

[Timmers 99] P. Timmers. Electronic Commerce: Strategies and Models for Business-to-Business Trading. JohnWiley & Sons Ltd., Chichester, England, 1999.

[Thomas 94] Thomas, I. “Software Processes and Business Processes,” Proceedings of the Third International Conference on the Software Processes: Applying the Software Process, Reston, VA,1994, p. 185

[Totland 95] T.Totland and R. Conradi, "A survey and comparison of some research area relevant to software process modelling". In Software Process Technology—Proceedings of the 4th European Software Process Modeling Workshop, W. Sch¨afer (Ed.), Noordwijkerhout, Netherlands: Springer, pp. 65–69. Appeared as Lecture Notes in Computer Science 913, 1995.

[Talvanen 02] Talvanen, J. P. "Domain Specific Modelling: Get your Products out 10 Times Faster". Real-Time & Embedded Computing Conference, 2002.

[UML 06] UML Virtual Machine Project, at http://dssg.cs.umb.edu/projects/umlvm.html, page last visit: May 27, 2007.

[Van Der Aalst 03a] Van der Aalst W.M.P, Ter Hofstede A.H.M., Kiepuszewski B., and Barros A.P. " Workflow Patterns", in journal of Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.

[Van Der Aalst 03b] Van der Aalst W.M.P, Ter Hofstede A.H.M., and Weske M., editors, International Conference on Business Process Management (BPM 2003), volume 2678 of Lecture Notes in Computer Science, pages 1-12. Springer-Verlag, Berlin, 2003.

[Van Gigch 91] van Gigch, J.P, "System Design Modeling and Metamodeling", Plenum Press, New York. ISBN 0-306-43740-6

[Vitolins 05] Vitolins V., Kalnins A., "Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine", in Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference, IEEE,2005:181-192

[Websphere] WebSphere Process Serve at http://www-1.ibm.com/support/docview.wss?uid=swg27007157&aid=1. Last time page visit 13/06/2007

[WFMC 06] WorkFlow Managment Coalition (WFMC), at http://www.wfmc.org/about.html, last time visited link: 12/26/2006

[WFMC 99] WorkFlow Managment Coalition (WFMC),Terminology & Glossary, Document Number WFMC-TC-1011, Feb. 1999

[WFMC 95] Reference Model - The Workflow Reference Model, WFMC-TC-1003, Jan 95, 1.1

[wikipidia] Wikipidia, Business Process, at http://en.wikipedia.org/wiki/Business_process

[Wise 00] Wise A., Cass A.G., Lerner B.S., McCall E.K., Osterweil L.J., Sutton, S.M., "Using Little-JIL to Coordinate Agents in Software Engineering", Proceedings of the Automated Software Engineering Conference (ASE 2000), Grenoble, France, pp. 155-163, September 2000

[WfP] Workflow Patterns at, http://www.workflowpatterns.com

[Wohed 04] Wohed P., van der Aalst W.M.P., Dumas M, ter Hofstede A.H.M., and Russel N., "Pattern-based Analysis of the Control-Flow Perspective of UML Activity Diagrams", in L. Delcambre et al., editors, Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005), volume 3716 of Lecture Notes in Computer Science, pages 63-78. Springer-Verlag, Berlin, 2005

[WSBEPL 07] Web Services Business Process Execution Language Version 2.0. Working Draft. WS-BPEL TC OASIS, January 2007. URL: http://www.oasis-open.org/committees/download.php/12791/

[WSDL 01] W3C Note, “Web Services Definition Language (WSDL) 1.1”, E.Christensen, F. Curbera, G. Meredith, S. Weerawarana, March 15, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[XML Schema 04a] “XML Schema Part 1: Structures Second Edition”, W3C Recommendation, H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, October 28, 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XML Schema 04b] “XML Schema Part 2: Datatypes Second Edition”, W3C Recommendation, P. V. Biron, A. Malhotra, October 28, 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[Xpath 99] “XML Path Language (XPath) Version 1.0”, W3C Recommendation, J.Clark, S. DeRose, November 1999. http://www.w3.org/TR/1999/RECxpath-19991116

[XSLT 99] “XSL Transformations (XSLT) Version 1.0”, W3C Recommendation, J. Clark, November 16, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116

[Zamli 01] K.Z. Zamli, , P.A. Lee, "Taxonomy of Process Modeling Languages". In: Proc. Of the ACS/IEEE International Conference on Computer Systems and Applications. IEEE Computer Society Press (June 2001) 435-437

[Zito 06a] Zito A., Diskin Z., and Dingel, J. "Package merge in UML 2: Practice vs. Theory" In Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), volume 4199 of LNCS, pages 185–199. Springer, 2006.

[Zito 06b] Zito A. and Dingel, J."Modeling UML2 package merge with Alloy". In First Alloy Workshop, Portland, Oregon, USA, November 2006.

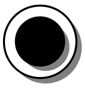
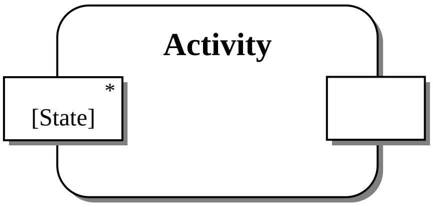
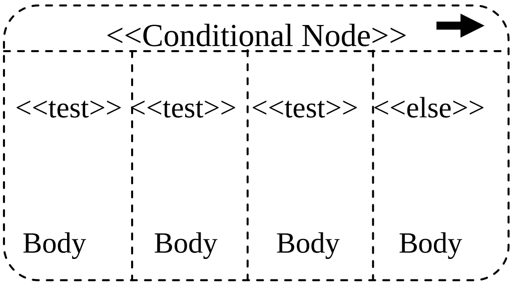
[2U 03] 2U Consortium. Unambiguous UML (2U) "3rd revised submission to UML 2 superstructure RFP". version 0.2. OMG document ad/2002-12-23, 2003

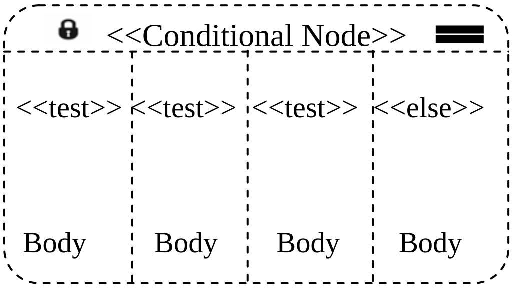
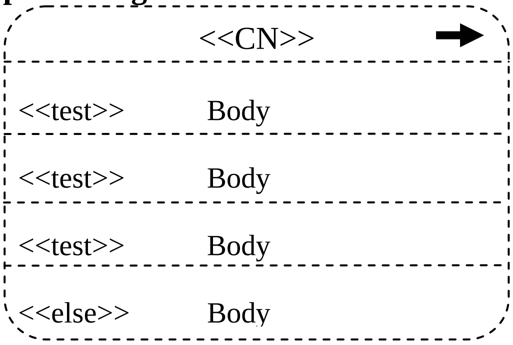
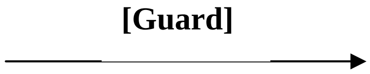
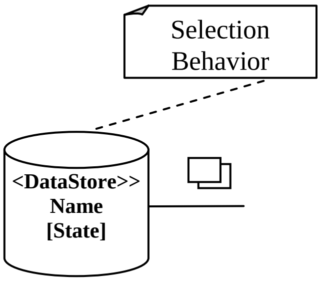
Appendix A

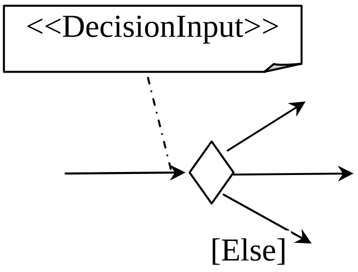
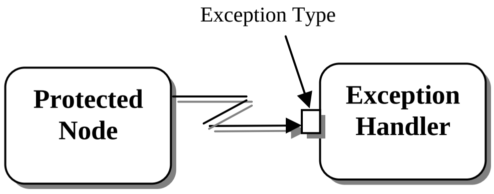
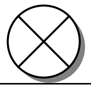
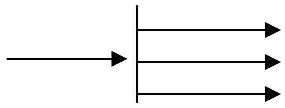
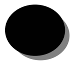
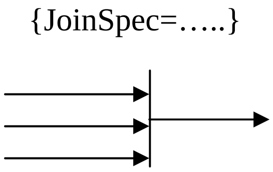
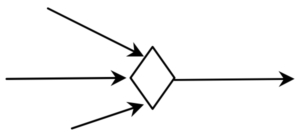
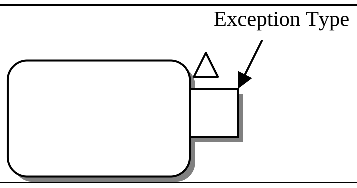
UML4SPM Notations

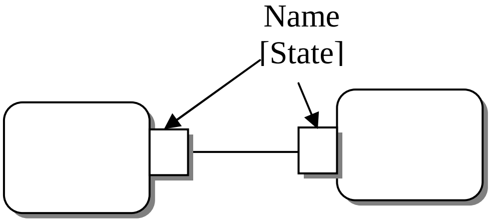
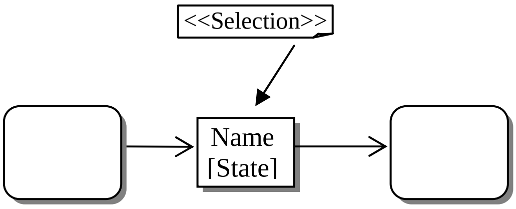
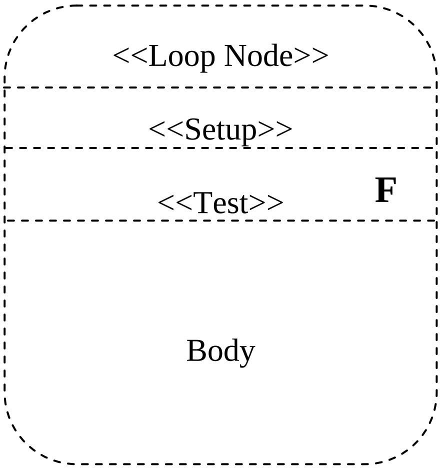
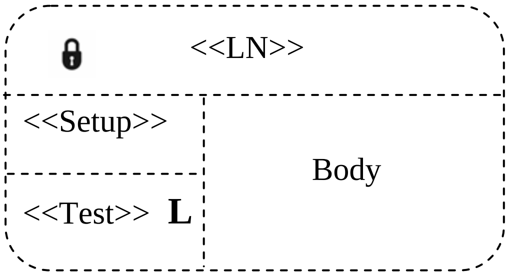
The notation of UML4SPM is mainly based on UML2.0 *Activity* notations. Some modifications were introduced in order to take into account some features proper to software process modeling but also to increase understandability. For UML2.0 *Activity* elements and *Actions* for which no notation is proposed by the standard, we proposed one. In this appendix, we present only the notation of elements we reused within UML4SPM.

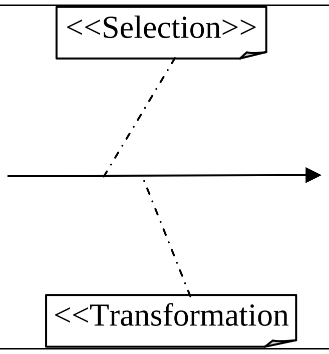
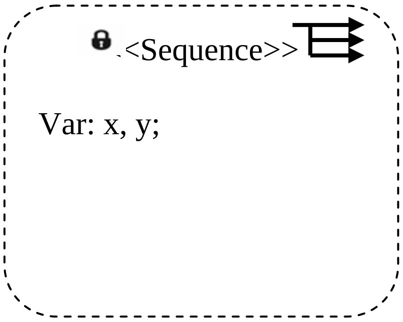
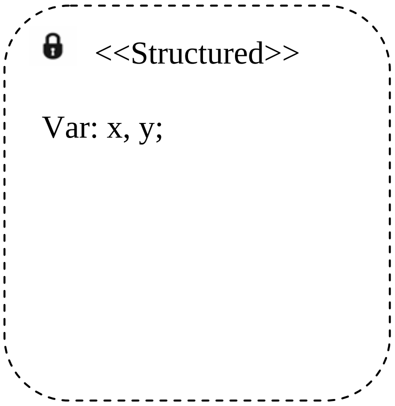
Activity Element Notations

Activity components : description	Notation
ActivityFinalNode (from BasicActivities, IntermediateActivities): an *Activity* may have more than one *ActivityFinalNode*. The first *ActivityFinalNode* that will be reached by a control flow will stop all *Activity*'s flows (stops action executions and ends the *Activity*).	
ActivityParameterNode (from BasicActivities): a specialization of *Object Node* which represents input and output parameters of an *Activity*. It is associated to a *Parameter* which has a *Type*. It is possible to define the multiplicity of the APN (e.g. "0") which makes it an optional or mandatory parameter for the *Activity* execution. We decided to enrich the notation by adding a star symbol (i.e., "*") to specify that the APN is optional. We can also specify the *State* of the parameter the *Activity* is receiving.	
ConditionalNode (from Structured Activities, CompleteStructuredActivities): is *StructuredActivityNode* and allows expressing a choice among many alternatives. *Conditional Nodes* (CN) are similar to *Decision Nodes* (defined below) but in a more structured way (equivalent to conditionals in programming languages). A CN contains *Clauses*; each *Clause* includes a *Test* and a *Body*. Only one clause is to be executed in case of more than one clause's test are evaluated at "true". It is also possible to have an "else" clause that will execute if all the other clauses are evaluated at "false".	**Conditional Node with a sequential evaluation:**  <<Conditional Node>> <<test>> <<test>> <<test>> <<else>> Body Body Body Body

A CD provides as an output, a *Result* which represents a set of *Outputpins*. Clauses can be evaluated in parallel or sequentially. The UML2.0 standard does not define a notation for this element. We propose a notation which takes as a basis, the one given for *StructuredActivityNodes*. An arrow symbol is used if the evaluation of the CN clauses is to be performed sequentially. Otherwise, "=" symbol is used in case of a parallel evaluation of CN clauses. For the expression of the clause's *Test* part, the use of an Action Language is encouraged for more readability (better than to use Activity elements and Actions to express the Test graphically). A Lock symbol is added on the top-left corner if the *mustIsolated* ="true". This means that variables handled within the CN are not accessible by actions which are outside the CN.	**Conditional Node with a parallel evaluation:**  <<Conditional Node>> <<test>> <<test>> <<test>> <<else>> Body Body Body Body **Conditional Node with horizontal portioning of clauses:**  <<CN>> <<test>> Body <<test>> Body <<test>> Body <<else>> Body
ControlFlow (from BasicActivities): expresses the passing of the control flow from one *Activity Node* into another. It is not possible to transfer objects or data through control flows. Being an *Activity Edge*, it is possible to specify *Guards* on control flows (conditions to be evaluated before passing the control flow).	
DataStoreNode (from CompleteAcitivities): A *DataStoreNode* keeps all objects (data) that enter it, copying them when they are chosen to move downstream. An incoming object replaces any occurrence of this object in the *DataStoreNode*. It is also possible to emit conditions upon which objects can go out of the *DataStoreNode* (e.g., state of the Object, the value of a property of the object, etc.). We modified the notation introduced by the standard which consisted in a simple rectangle to a cylinder -shaped form which is more common for representing storage entities.	

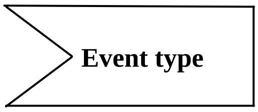
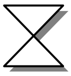
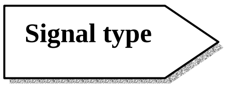
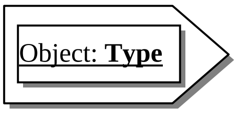
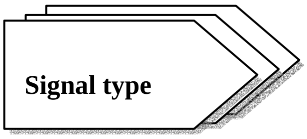
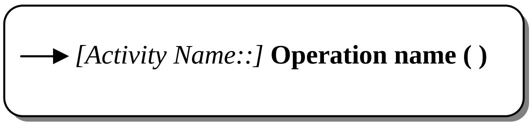
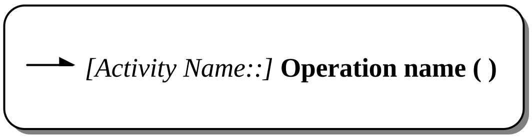
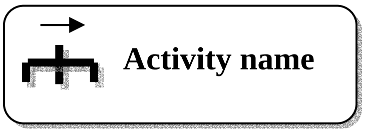
DecisionNode (IntermediateActivities): A *DecisionNode* has one incoming edge and multiple outgoing activity edges with *Guards*. It is possible to define an "else" outgoing edge that will be triggered in case of all outgoing edge guards are evaluated at "false". It is also possible to define a *Decision Input behavior* that will for instance extract an object's property value in order to pass it to outgoing activity edge *Guards* for evaluation instead of passing the object.	
ExceptionHandler (ExtraStructuredActivities): allows the specification of a behavior to execute in case of an exception occurs. The *ExceptionHandler* is triggered by a *RaiseExceptionAction* which specifies the exception type. When the exception is raised, the node protected by the handler is stopped	
FlowFinalNode (IntermediateActivities): stops all incoming flows. Does not affect the other flows.	
ForkNode (IntermediateActivities): A *ForkNode* is a control node that splits a flow into multiple concurrent flows. Combined with a *CallBehaviorAction*, this would allow calling multiple activities simultaneously.	
InialNode (BasicActivities): An *InitialNode* is a starting point for executing an *Activity*. An *Activity* may have many *InitialNodes* and when calling the *Activity*, all its *InitialNodes* will be activated.	
JoinNode (IntermediateActivities): A *JoinNode* is a control node that synchronizes multiple flows. It has multiple incoming edges and one outgoing edge. A specification giving the conditions under which the join will be activated can be express thanks to the *JoinSpec* property.	
MegeNode (IntermediateActivites): A *MergeNode* is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows. Usually, it is used after a *DecisionNode*.	
Pin (BasicActivities, CompleteActivities) : represents input (Input Pins) and outputs (OutputPons) of actions. Several notations are proposed.	

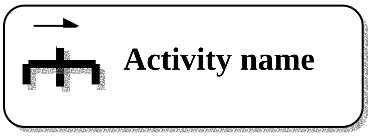
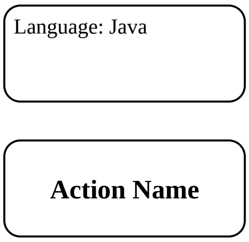
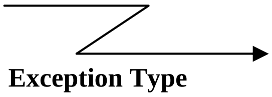
	Or:  **Or:** 
LoopNode (CompleteStructuredActivitie, StructuredActivities): is a *StructuredActivityNode*, which allows expressing loops as in programming languages. A loop includes a *Setup*, a *Test* and a *Body* that might be described using *Activity Nodes*. The *Setup* part is executed first. The *Test* part can be executed either before the *Body* part or after depending on the value of the ***isTestedFirst*** property. The UML2.0 does not define a notation for the LoopNode. We propose a notation which takes as a basis, the one given for *StructuredActivityNodes*. For the expression the *Test* part of the clause, the use of an Action Language is encouraged for more readability (better than to use Activity elements and Action to express the Test graphically). An "F" or "L" character is used to specify if the *Test* part is executed before the *Body* (i.e. First "F") or after (i.e., Last "L"). A Lock symbol is added on the top-left corner if the *mustIsolated* ="true". This means that variables handled within the CN are not accessible by actions which are outside the CN.	**Loop Node:**  **Or:** 
ObjectFlow (BasicActivities, CompleteActivities): An *ObjectFlow* is an activity edge that can have objects or data passing along it. It is possible to apply a behavior upon the data passing the edge before passing it to the target node (thanks to	

the *transformation* association). This behavior must not modify the object. It also possible to specify a selection (e.g. only objects with property state="validated", etc.). This is done using the *selection* association.	
SequenceNode (StructuredActivities): allows structuring a set of ordered executable nodes. The UML2.0 does not define a notation for the *SequenceNode*. We propose a notation which takes as a basis, the one given for *StructuredActivityNodes*. A multiple arrow symbol is used to differentiate with the notation given for the *StructuredActivityNode*. A Lock symbol is added on the top-left corner if the *mustIsolated* ="true". This means that variables handled within the CN are not accessible by actions which are outside the CN.	
StructureActivityNode (CompleteStructuredActivities, StrcuturedActvities): A *StructuredActivityNode* represents a structured portion of the *Activity* that is not shared with any other structured node, except for nesting. It may have control edges connected to it, and pins. The execution of any embedded actions may not begin until the *StructuredActivityNode* has received its entire object and control flows. The availability of output pins from the *StructuredActivityNode* does not occur until all embedded actions have completed execution. No data manipulated inside the *StructuredActivityNode* is reachable from the outside if the *mustIsolated* property is set to "true". The standard proposes the following notation. We decided to add the lock symbol to state if the *mustIsolated* is set to "true". We will extend this notation for representing specializations of the *StructuredActivityNode*	

Action Notations

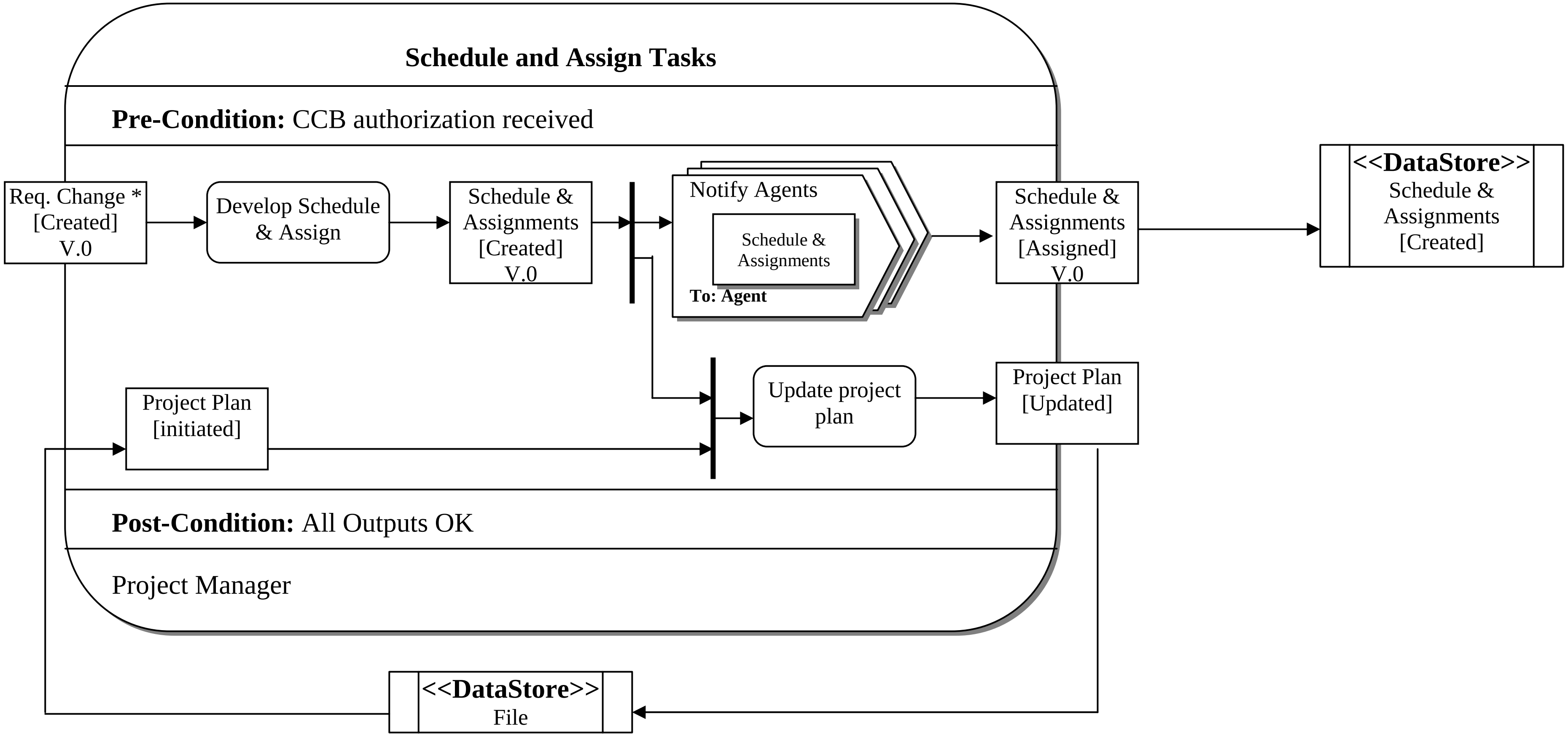
In this section, we present action notations. The description of actions retained in UML4SPM was given in the previous chapter (cf. Section 3.2.3. UML2.0 Actions reused within UML4SPM).

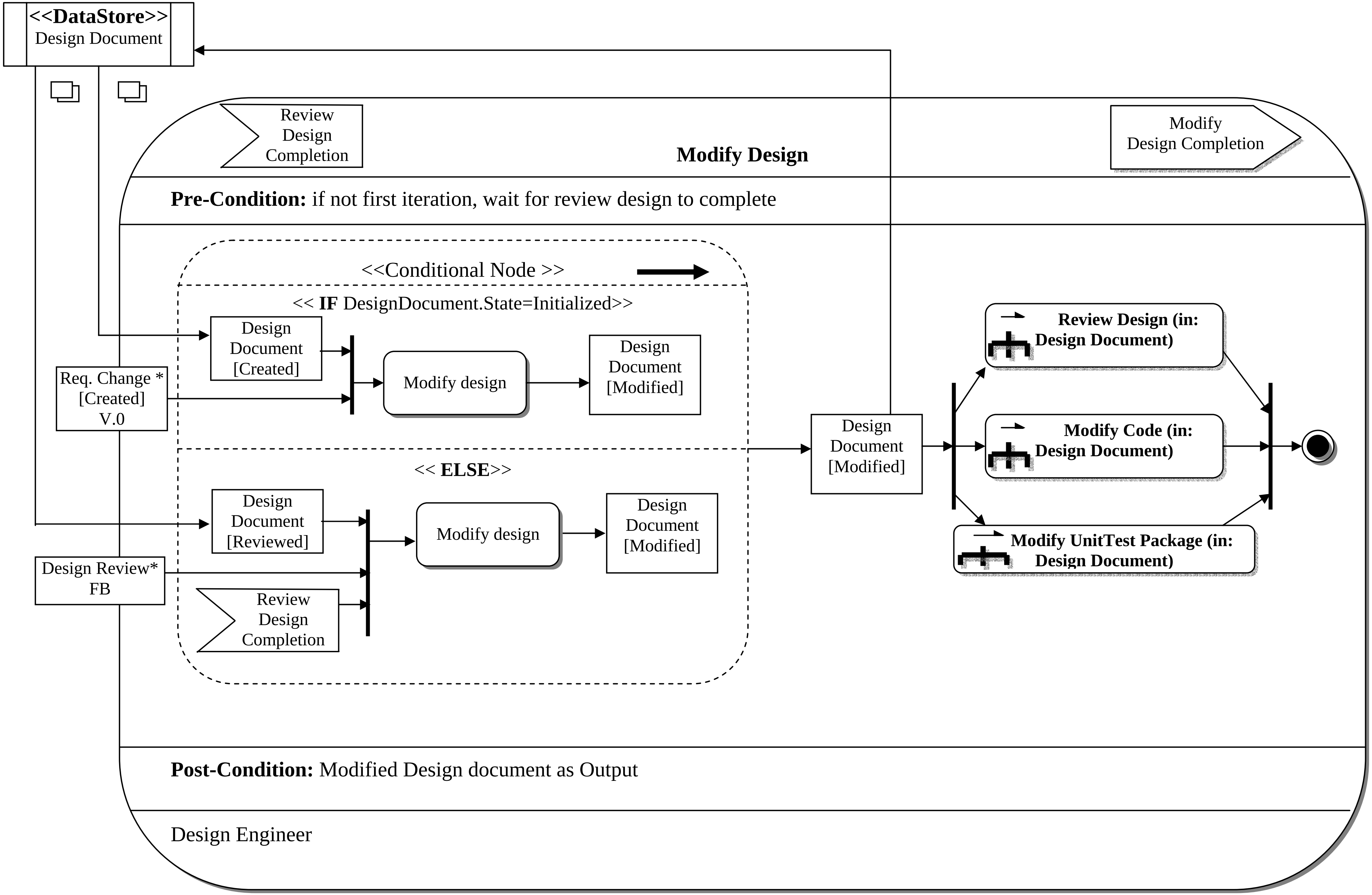
Actions	Notations
AcceptEvenAction (Complete Actions)	***- For all types of event except Time Event:***  ***- Only for Time event:*** 
SendSignalAction (Basic Actions)	
SendObjectAction (from IntermediateActions): the standard does not define a notation for this action. We propose one.	
BroadcastSignalAction (from IntermediateActions): the standard does not define a notation for this action. We propose one.	
CallOperationAction (form BasicActions): We decided to enrich the notation with the arrow symbol in order to specify if the call is synchronous (full arrow) or asynchronous (half arrow). The Activity name is optional in case of the C*allOperationAction* and the called *Operation* are owned by the same Activity	**Synchronous Call**  **Asynchronous Call** 
CallBehaviorAction (from BasicActions): As for the C*allOperationAction*, we decided to enrich the notation with the arrow symbol in order to specify if the call is	**Synchronous Call** 

synchronous (full arrow) or asynchronous (half arrow).	**Asynchronous Call** 
OpaqueAction (from BasicActions): the standard does not define a specific notation for this action. We reuse the same notation for actions and we add the possibility to document the language of the opaque action (e.g. Java). In absence of the language property, the name of the action is used to describe the intension of the action (e.g. IdentifyAssociations). In UML4SPM Opaque Action is used to model manual actions and human interactions.	
RaiseExceptionAction (Structured Actions) : the standard does not define a notation for this action. We propose one.	

ISPW-6 Process Example modeled using UML4SPM

In this Appendix, we give only the result of modeling the ISPW-6 Process example using the UML4SPM SPML. The details of what has to be performed within each activity, their sequencing, their inputs, outputs, roles, and constraints are given in [Kellner 91b].


Schedule and Assign Tasks
Pre-Condition: CCB authorization received
Req. Change * [Created] V.0
Develop Schedule & Assign
Schedule & Assignments [Created] V.0
Notify Agents
Schedule & Assignments
To: Agent
Schedule & Assignments [Assigned] V.0
<<DataStore>> Schedule & Assignments [Created]
Project Plan [initiated]
Update project plan
Project Plan [Updated]
Post-Condition: All Outputs OK
Project Manager
<<DataStore>> File



<<DataStore>>
Design Document

Review Design Completion

Modify Design

Modify Design Completion

Pre-Condition: if not first iteration, wait for review design to complete

<<Conditional Node >>

<< **IF** DesignDocument.State=Initialized>>

Design Document [Created]

Req. Change * [Created] V.0

Modify design

Design Document [Modified]

<< **ELSE**>>

Design Document [Reviewed]

Design Review* FB

Review Design Completion

Modify design

Design Document [Modified]

Design Document [Modified]

Review Design (in: Design Document)

Modify Code (in: Design Document)

Modify UnitTest Package (in: Design Document)

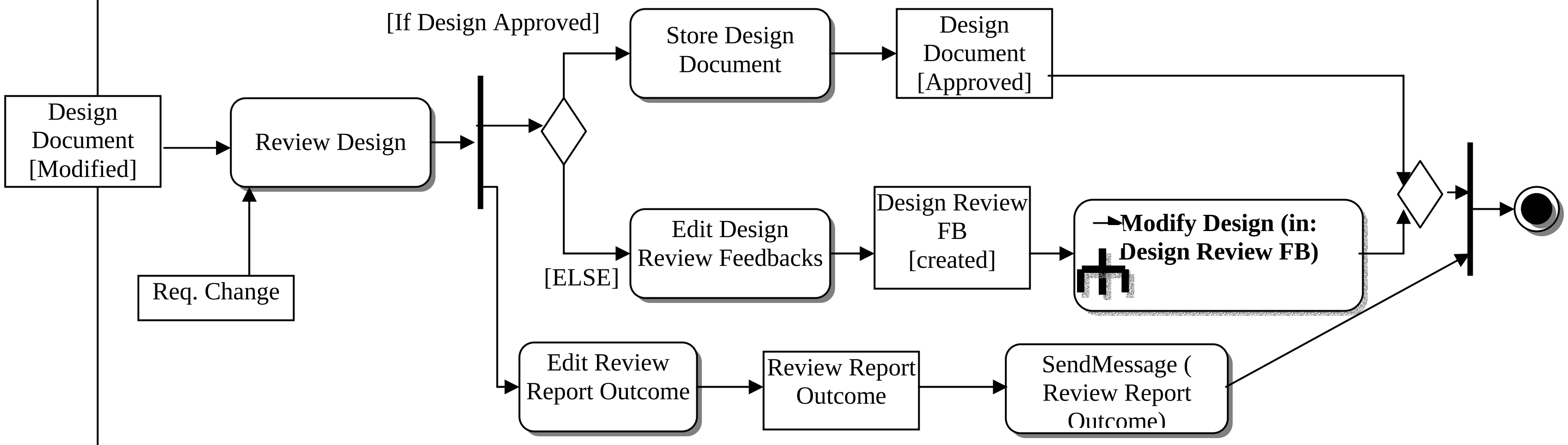
Post-Condition: Modified Design document as Output

Design Engineer

Review Design

Review
Design Completion

Pre-Condition: Design Documents Modified available, Verbal authorization form CCB



Design Document [Modified]

Review Design

Req. Change

[If Design Approved]

Store Design Document

Design Document [Approved]

[ELSE]

Edit Design Review Feedbacks

Design Review FB [created]

Modify Design (in: Design Review FB)

Edit Review Report Outcome

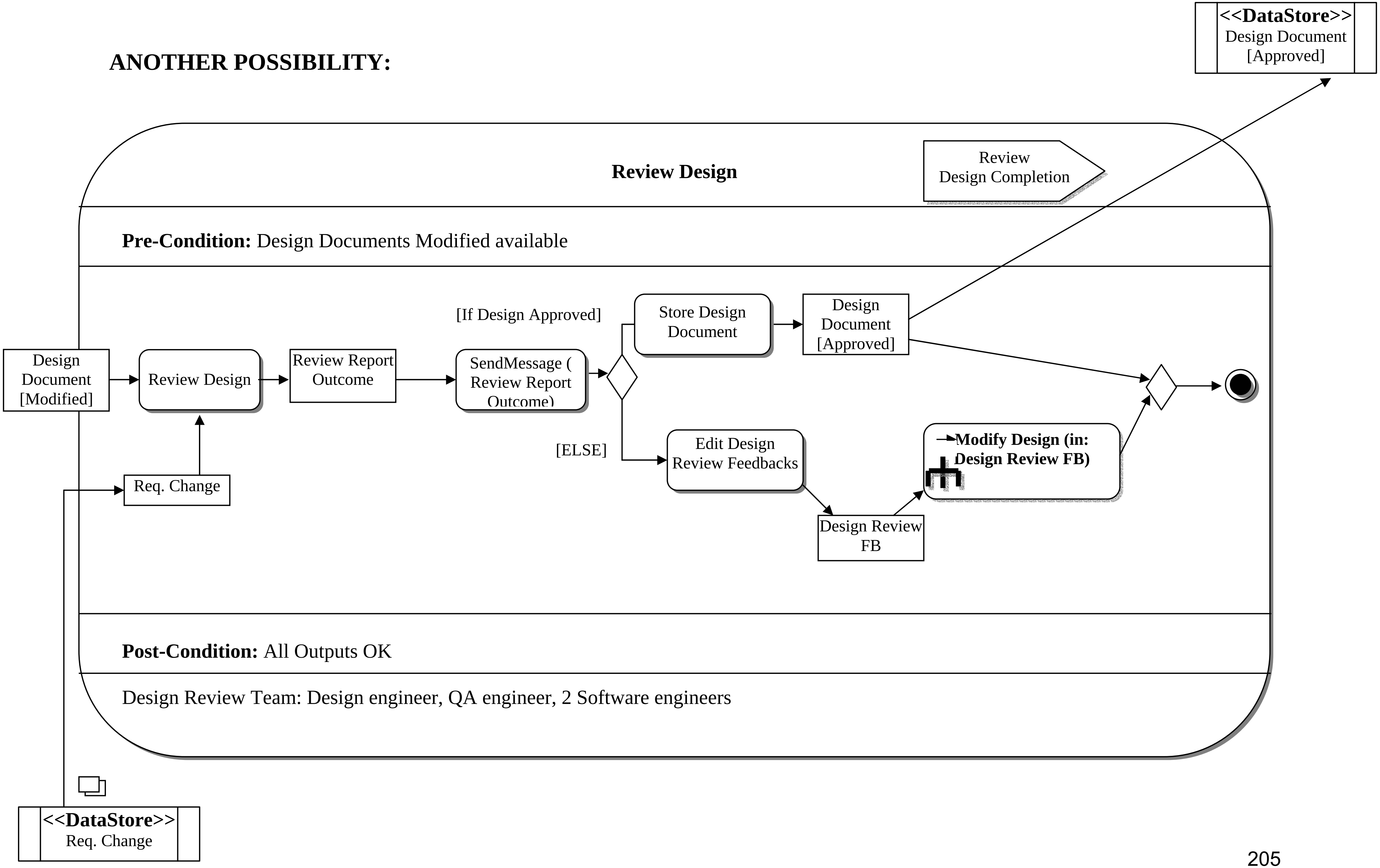
Review Report Outcome

SendMessage (Review Report Outcome)

Post-Condition: All Outputs OK

Design Review Team: Design engineer, QA engineer, 2 Software engineers

ANOTHER POSSIBILITY:



<<**DataStore**>>
Design Document
[Approved]

Review Design

Review
Design Completion

Pre-Condition: Design Documents Modified available

Design Document [Modified]

Review Design

Review Report Outcome

SendMessage (Review Report Outcome)

[If Design Approved]

Store Design Document

Design Document [Approved]

[ELSE]

Edit Design Review Feedbacks

Design Review FB

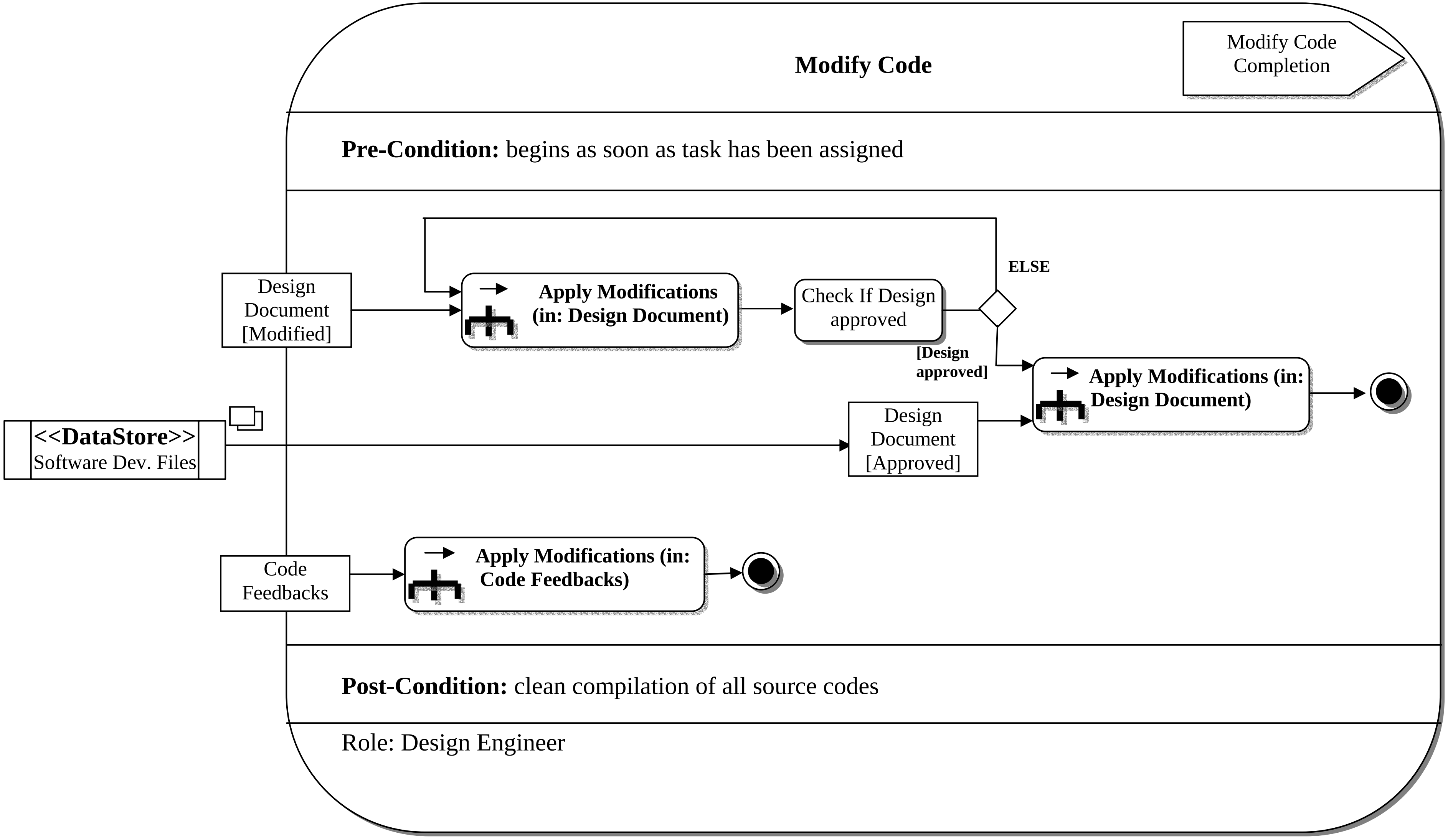
Modify Design (in: Design Review FB)

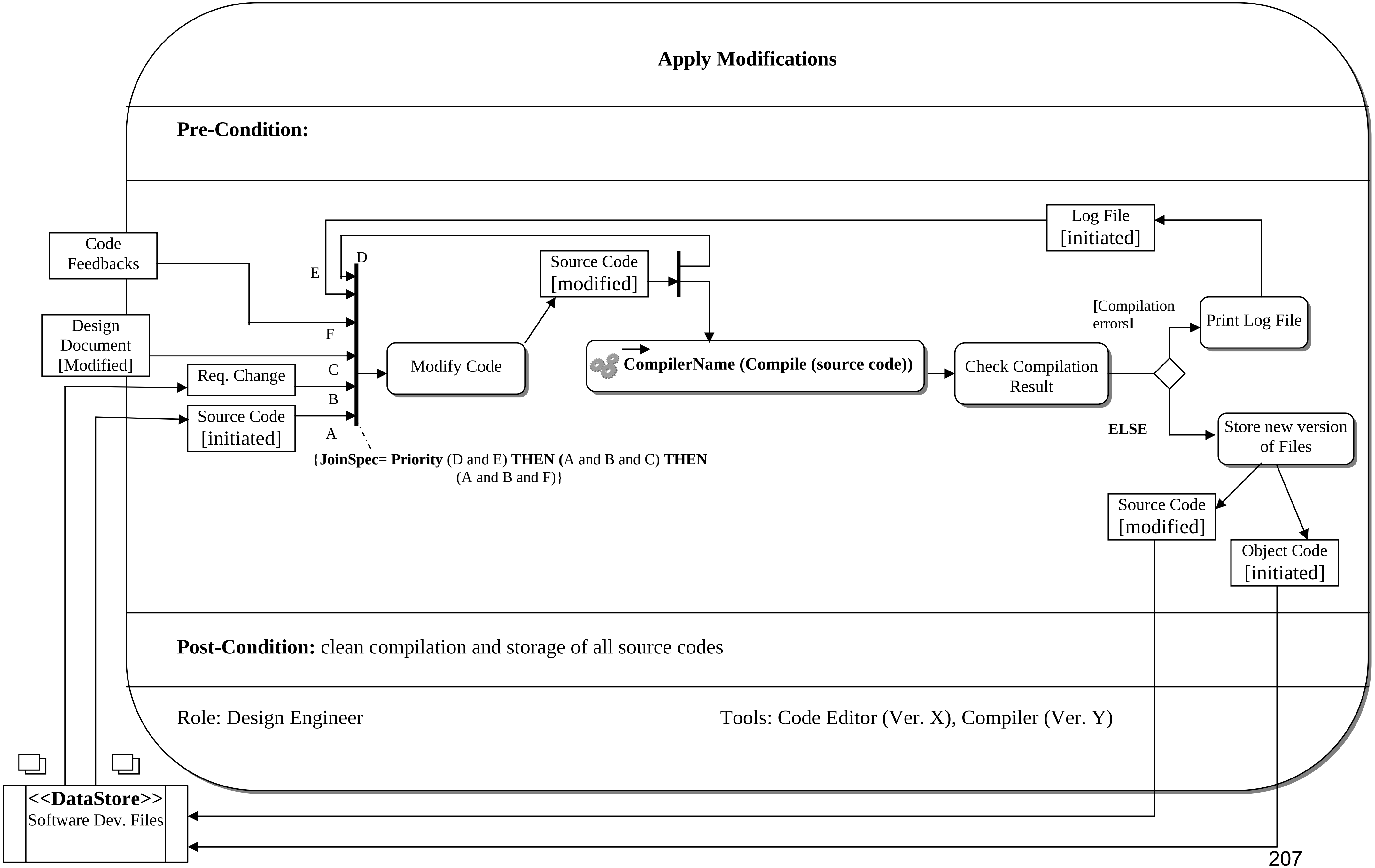
Req. Change

Post-Condition: All Outputs OK

Design Review Team: Design engineer, QA engineer, 2 Software engineers

<<**DataStore**>>
Req. Change

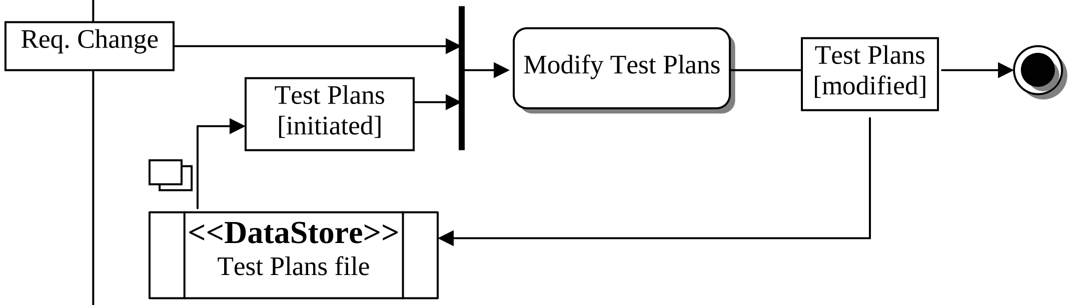

Modify Code
Modify Code Completion
Pre-Condition: begins as soon as task has been assigned
Design Document [Modified]
Apply Modifications (in: Design Document)
Check If Design approved
ELSE
[Design approved]
Apply Modifications (in: Design Document)
<<DataStore>> Software Dev. Files
Design Document [Approved]
Code Feedbacks
Apply Modifications (in: Code Feedbacks)
Post-Condition: clean compilation of all source codes
Role: Design Engineer


Apply Modifications
Pre-Condition:
Code Feedbacks
Design Document [Modified]
Req. Change
Source Code [initiated]
E
D
F
C
B
A
{JoinSpec= Priority (D and E) THEN (A and B and C) THEN (A and B and F)}
Modify Code
Source Code [modified]
Log File [initiated]
CompilerName (Compile (source code))
Check Compilation Result
[Compilation errors]
Print Log File
ELSE
Store new version of Files
Source Code [modified]
Object Code [initiated]
Post-Condition: clean compilation and storage of all source codes
Role: Design Engineer
Tools: Code Editor (Ver. X), Compiler (Ver. Y)
<<DataStore>>
Software Dev. Files

Modify Test Plans

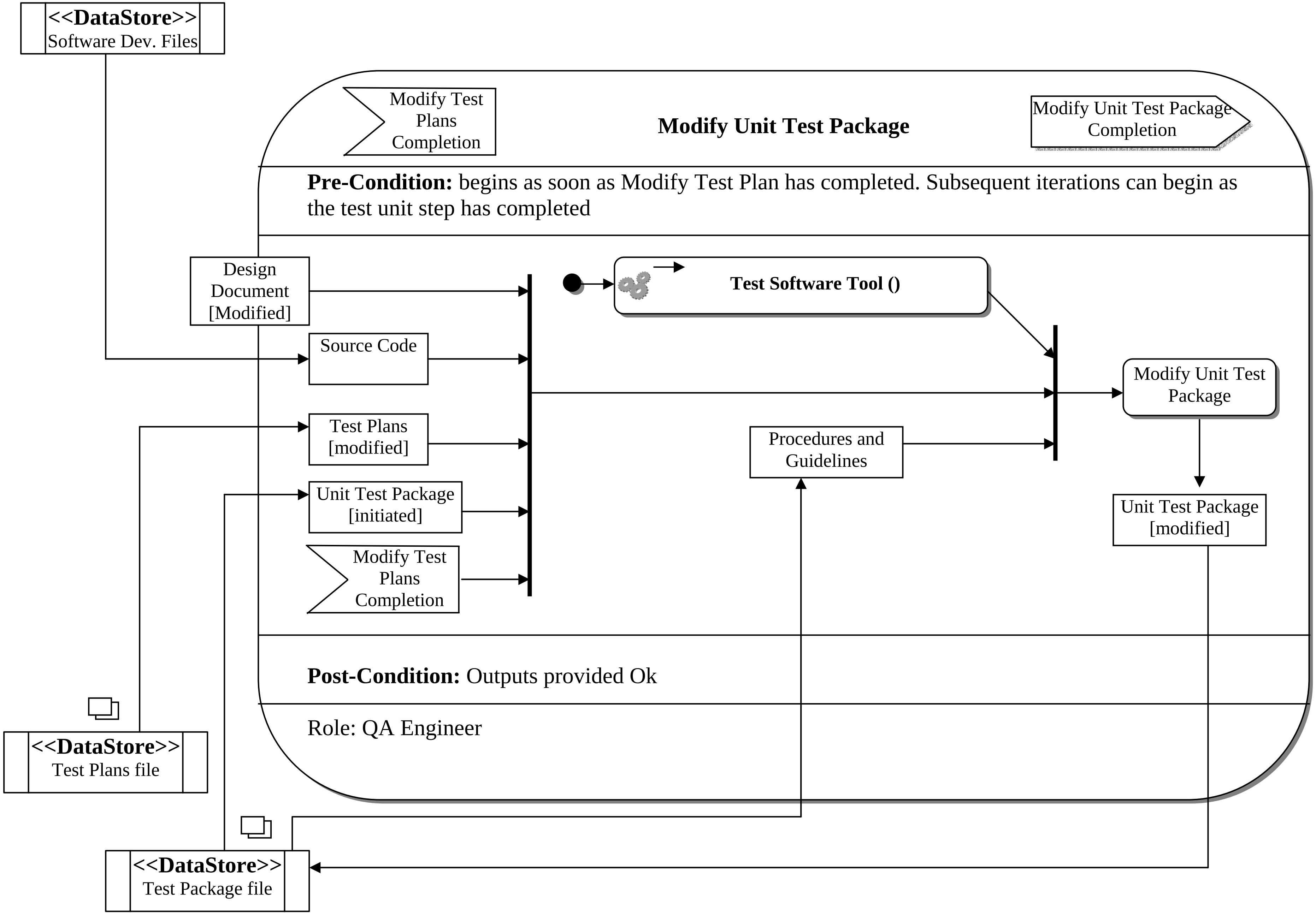
Modify Test Plans Completion

Pre-Condition: begins as soon as task has been assigned by the Project Manager



Post-Condition: Outputs provided Ok

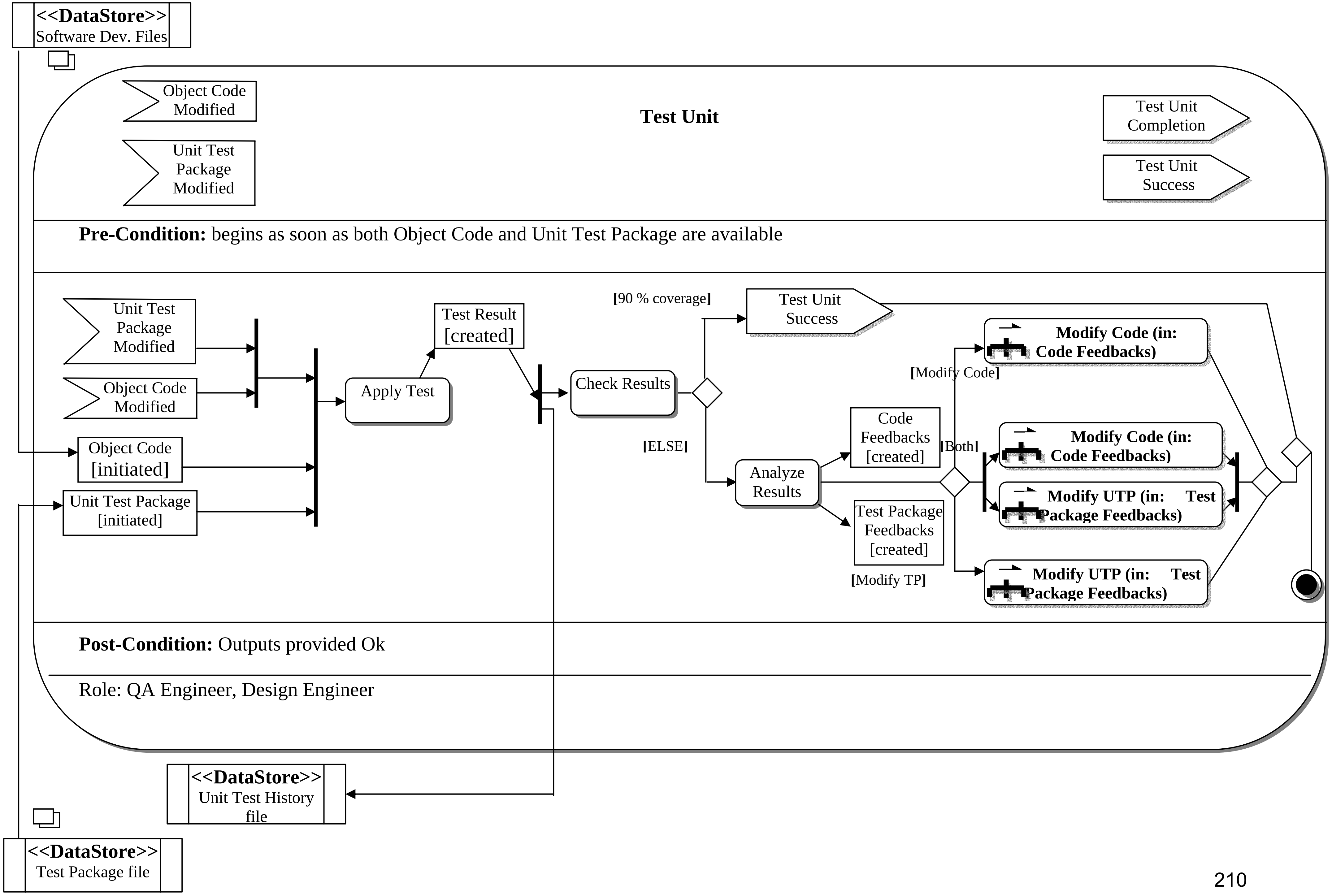
Role: Design Engineer

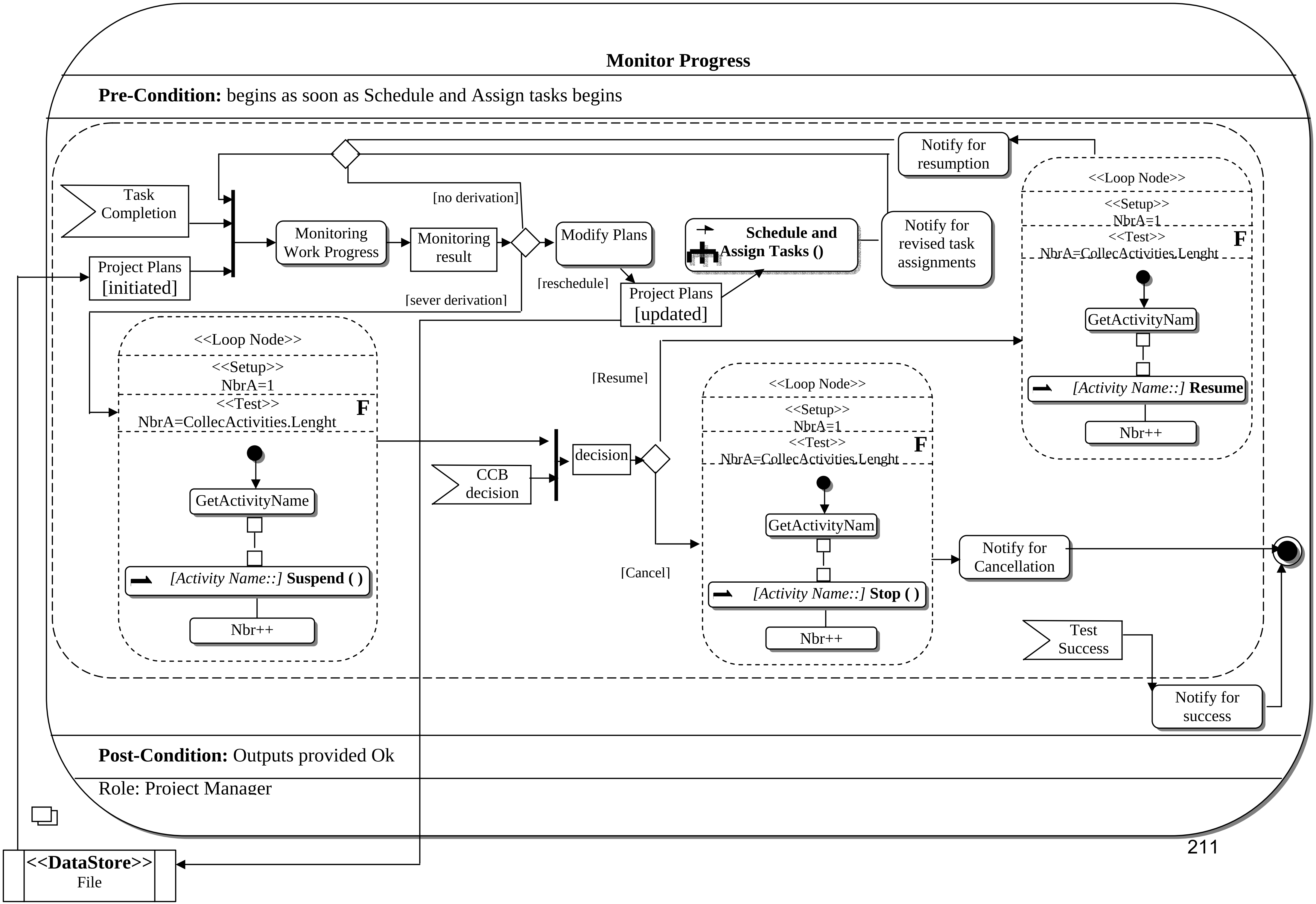

<<DataStore>>
Software Dev. Files
Modify Test Plans Completion
Modify Unit Test Package
Modify Unit Test Package Completion
Design Document [Modified]
Test Software Tool ()
Source Code
Modify Unit Test Package
Test Plans [modified]
Procedures and Guidelines
Unit Test Package [initiated]
Unit Test Package [modified]
Modify Test Plans Completion
<<DataStore>>
Test Plans file
<<DataStore>>
Test Package file

Pre-Condition: begins as soon as Modify Test Plan has completed. Subsequent iterations can begin as the test unit step has completed

Post-Condition: Outputs provided Ok

Role: QA Engineer


<<DataStore>>
Software Dev. Files
Object Code Modified
Unit Test Package Modified
Test Unit
Test Unit Completion
Test Unit Success
Pre-Condition: begins as soon as both Object Code and Unit Test Package are available
Unit Test Package Modified
Object Code Modified
Object Code [initiated]
Unit Test Package [initiated]
Apply Test
Test Result [created]
Check Results
[90 % coverage]
Test Unit Success
[ELSE]
Analyze Results
Code Feedbacks [created]
Test Package Feedbacks [created]
[Modify Code]
[Both]
[Modify TP]
Modify Code (in: Code Feedbacks)
Modify Code (in: Code Feedbacks)
Modify UTP (in: Test Package Feedbacks)
Modify UTP (in: Test Package Feedbacks)
Post-Condition: Outputs provided Ok
Role: QA Engineer, Design Engineer
<<DataStore>>
Unit Test History file
<<DataStore>>
Test Package file


Monitor Progress
Pre-Condition: begins as soon as Schedule and Assign tasks begins
Task Completion
Project Plans [initiated]
Monitoring Work Progress
Monitoring result
[no derivation]
[sever derivation]
Modify Plans
[reschedule]
Project Plans [updated]
Schedule and Assign Tasks ()
Notify for revised task assignments
Notify for resumption
<<Loop Node>>
<<Setup>>
NbrA=1
<<Test>>
NbrA=CollecActivities.Lenght
F
GetActivityNam
[Activity Name::] Resume
Nbr++
[Resume]
<<Loop Node>>
<<Setup>>
NbrA=1
<<Test>>
NbrA=CollecActivities.Lenght
F
GetActivityName
[Activity Name::] Suspend ( )
Nbr++
CCB decision
decision
<<Loop Node>>
<<Setup>>
NbrA=1
<<Test>>
NbrA=CollecActivities.Lenght
F
GetActivityNam
[Activity Name::] Stop ( )
Nbr++
[Cancel]
Notify for Cancellation
Test Success
Notify for success
Post-Condition: Outputs provided Ok
Role: Project Manager
<<DataStore>>
File

Appendix B

Listing 1. WS-BPEL sample of the Inception Phase

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveBPEL(tm) Designer Version 3.0.0 (http://www.active-
endpoints.com)
-->
<bpel:process xmlns:bpel="http://docs.oasis-
open.org/wsbpel/2.0/process/executable"
xmlns:ns1="http://www.softeam.fr/WorkflowAdministration/"
xmlns:ns2="http://www.example.org/orchestration/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Inception"
suppressJoinFailure="yes" targetNamespace="http://Inception">
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/"
location="WorkflowAdministration.wsdl"
namespace="http://www.softeam.fr/WorkflowAdministration/"/>
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/"
location="../orchestration/orchestration.wsdl"
namespace="http://www.example.org/orchestration/"/>
   <bpel:partnerLinks>
      <bpel:partnerLink myRole="HumanActivityFacade"
name="HumanActivity" partnerLinkType="ns1:HumanActivity"
partnerRole="HumanActivityFacade"/>
      <bpel:partnerLink name="OrchTool" partnerLinkType="ns2:OrchTool"
partnerRole="OrchastrationProvider"/>
   </bpel:partnerLinks>
   <bpel:variables>
      <bpel:variable messageType="ns1:HumanActivityRequest"
name="InceptionRequest"/>
      <bpel:variable messageType="ns1:HumanActivityRequest"
name="ElaborateAnalysisModelRequest"/>
      <bpel:variable messageType="ns1:HumanActivityResponse"
name="ElaborateAnalysisModelResponse"/>
      <bpel:variable messageType="ns1:HumanActivityRequest"
name="ValidateAnalysisModelRequest"/>
      <bpel:variable messageType="ns1:HumanActivityResponse"
name="ValidateAnalysisModelResponse"/>
      <bpel:variable messageType="ns2:sendMailRequest"
name="sendMailRequest"/>
   </bpel:variables>
   <bpel:flow>
      <bpel:links>
         <bpel:link name="L1"/>
         <bpel:link name="L2"/>
         <bpel:link name="L3"/>
         <bpel:link name="L4"/>
         <bpel:link name="L5"/>
      </bpel:links>
      <bpel:receive createInstance="yes" name="StartInception"
operation="HumanActivityRequest" partnerLink="HumanActivity"
portType="ns1:WorkflowAdministrationPT" variable="InceptionRequest">
         <bpel:sources>
            <bpel:source linkName="L1"/>
         </bpel:sources>
      </bpel:receive>
      <bpel:invoke inputVariable="ElaborateAnalysisModelRequest"
name="ElaborateAnalysisModelRequest" operation="HumanActivityRequest"
partnerLink="HumanActivity" portType="ns1:WorkflowAdministrationPT">
         <bpel:targets>
            <bpel:target linkName="L1"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L2"/>
         </bpel:sources>
      </bpel:invoke>
      <bpel:receive name="ElaborateAnalysisModelResponse"
operation="HumanActivityResponse" partnerLink="HumanActivity"
portType="ns1:WorkflowAdministrationPT"
variable="ElaborateAnalysisModelResponse">
         <bpel:targets>
            <bpel:target linkName="L2"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L3"/>
         </bpel:sources>
      </bpel:receive>
      <bpel:invoke inputVariable="ValidateAnalysisModelRequest"
name="ValidateAnalysisModelRequest" operation="HumanActivityRequest"
partnerLink="HumanActivity" portType="ns1:WorkflowAdministrationPT">
         <bpel:targets>
            <bpel:target linkName="L3"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L4"/>
         </bpel:sources>
      </bpel:invoke>
      <bpel:receive name="ValidateAnalysisModelResponse"
operation="HumanActivityResponse" partnerLink="HumanActivity"
portType="ns1:WorkflowAdministrationPT"
variable="ValidateAnalysisModelResponse">
         <bpel:targets>
```

```
            <bpel:target linkName="L4"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L5"/>
         </bpel:sources>
      </bpel:receive>
      <bpel:if>
         <bpel:targets>
            <bpel:target linkName="L5"/>
         </bpel:targets>
         <bpel:condition
expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">$Val
idateAnalysisModelResponse.stringResult='true'</bpel:condition>
         <bpel:invoke                     inputVariable="sendMailRequest"
name="SendMessageOk"     operation="sendMail"     partnerLink="OrchTool"
portType="ns2:orchestration"/>
         <bpel:else>
            <bpel:invoke                  inputVariable="sendMailRequest"
name="SendMessageFailure"   operation="sendMail"   partnerLink="OrchTool"
portType="ns2:orchestration"/>
         </bpel:else>
      </bpel:if>
   </bpel:flow>
</bpel:process>
```

Appendix C

ProcessModelExecution class

```
package ExecActivity;
import move.lip6.uml4spm.uml4spm.*;
import move.lip6.uml4spm.uml4spm.impl.UML4SPMPackageImpl;
.....
.....


public class ProcessModelExecution {


//a Hash Tab containing software activities and their corresponding execution
//activities
public  static Hashtable <SoftwareActivity, ActivityExecution>
SoftwareActivitiesMap=new Hashtable <SoftwareActivity, ActivityExecution>();

//a Hash Tab containing WorkProducts, we use the name as Key
public static Hashtable <String, WorkProduct> workProductList=new
Hashtable<String, WorkProduct>();

//a Hash Tab containing Responsible Role, we use the name as Key
public static Hashtable <String, ResponsibleRole> responsibleRoleList =new
Hashtable <String, ResponsibleRole>();

//Keep a ref to the Initial Activity Execution (i.e., its isInitial property
//equals at true).It will be the first to be executed and represents the context
//of the process
   public static ActivityExecution initialSoftwareActivity=null;

//Software Activities -Exections-
public static List <ActivityExecution> softwareActivitiesExecution=new Vector
<ActivityExecution>();
.....
.....
.....


public static void ActivityExecutionFactory(Collection activities){

    if (activities!=null){

           Iterator it = activities.iterator();
           SoftwareActivity act=null;
            while(it.hasNext()) {
               act=(SoftwareActivity)it.next();
                System.out.println("The Software Activity name is :" +
                act.getName());

               //For each softwareActivity creates its equivalent executable SA
                ActivityExecution actExec = new ActivityExecution(act);

                //Inialize the SA exectuion i.e., creates it nodes (actions,
                //control nodes, edges, object nodes)
                actExec.intialize();

                //check if the Activity is the initial one or not
                if (act.isInitial())
                      initialSoftwareActivity=actExec;

                //Keep a link between a SA definition and its execution instance
                SoftwareActivitiesMap.put((SoftwareActivity)act,
                (ActivityExecution)actExec);

                //save the activity executions of the process
                softwareActivitiesExecution.add(actExec);
              }
       }
       else
       {
         System.out.println("Package Empty!!!, no activities!!");
       }

   }

//method for adding a new responsible role to the process
public static void setResponsibleRoles(Collection <ResponsibleRole> respRoles){
          if (!(respRoles.isEmpty())){
                 Iterator it=respRoles.iterator();
                 while (it.hasNext()){
                       ResponsibleRole rspR=(ResponsibleRole)it.next();
                       addResponsibleRole(rspR);
                 }
          }
          else
                 System.out.println("Collection of Responsible Roles is
Empty!!!!!");
   }

public static void  addResponsibleRole(ResponsibleRole rspR){
          responsibleRoleList.put(rspR.getName(), rspR);

   }

//method for deleting a new responsible role to the process
public static void  deleteResponsibleRole(){
          ......

   }

//method for adding a new workproduct to the process
public static void setWorkProducts(Collection <WorkProduct> workproducts){
          if (workproducts!=null){
                 Iterator it=workproducts.iterator();
                 while (it.hasNext()){
                       WorkProduct workP=(WorkProduct)it.next();
                       addWorkProduct(workP);
                 }
          }
          else
                 System.out.println("Collection of Responsible Roles is
                 Empty!!!!!");
   }
public static void  addWorkProduct(WorkProduct workP){
          workProductList.put(workP.getName(), workP);
   }

//method for deleting a new responsible role to the process
   public static void  deleteWorkProduct(){
          ......

   }

//the main of the process execution
public static void main(String[] args) {
          // TODO Auto-generated method stub

          System.out.println("Program Start");

          // A step in order to make UML4SPM Metamodel ready to be referenced
          UML4SPMPackageImpl.init();

          // a set of steps in order to load the process model in memory
          URI fileURI =
          URI.createFileURI("C:\\eclipse\\workspace\\SoftwareProcessExample\\Sof
          twareProcessExample.uml4spm");

          Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put("uml
          4spm", new XMIResourceFactoryImpl(){
                    public Resource createResource(URI uri) {
                      XMIResource xmiResource = new XMIResourceImpl(uri);
                      return xmiResource;
                    }
                  });

          ResourceSet rs = new ResourceSetImpl();

          //Create a Resource in order to manipulate the process model instance
          Resource resource = rs.getResource(fileURI, true);

          //Get the outermost element of the process model which is the "Process
          //Model" element

          move.lip6.uml4spm.uml4spm.ProcessModel myModel =
          (move.lip6.uml4spm.uml4spm.ProcessModel)
          EcoreUtil.getObjectByType(resource.getContents(),move.lip6.uml4spm.uml
          4spm.UML4SPMPackage.eINSTANCE.getProcessModel());
                    System.out.println(myModel.getName());

          //get all package elements
          Collection ProcessElments = myModel.getProcessElements();

          //get WorkProducts used whithin the process
          Collection processWorkProducts=
          findElementByType(ProcessElments,"WorkProduct");
          setWorkProducts(processWorkProducts);

          //get Responsbile Roles needed within the process
          Collection processResponsibleRoles=
          findElementByType(ProcessElments,"ResponsibleRole");
          setResponsibleRoles(processResponsibleRoles);

          //get process model activities
          Collection processModelActivities=
          findElementByType(ProcessElments,"SoftwareActivity");

          //Call the ActivityExecutionFactory method, for each activity in the
          //set, create its equivalent activityExecution
          ActivityExecutionFactory(processModelActivities);

          //launch the execution of the process by called the execute method on
          //its initial software Activity
          if (initialSoftwareActivity!=null)
                initialSoftwareActivity.execute();

          //in case that no isIntial activity's property in the process model is
          //set to true
          else {
                System.out.println("Process Model must have one Initial Software
                Activity (i.e.) its isInitial property=true");
                if (!(softwareActivitiesExecution.isEmpty())){
                //in case of only one actvity than execute it otherwise, process
                //model can't be executed
                if (softwareActivitiesExecution.size()==1){
                      softwareActivitiesExecution.get(0).execute();
                }
                else
                      System.out.println("Process Model contains more than one
                      activity with no one with its attribute 'isInitial' set
                      at 'true'");

          }
          else
                System.out.println("Process Model does not contain anay
                activity");
       }


       System.out.println("Execution End ");
   }
}
```

ActivityExecution class

```
package ExecActivity;

import move.lip6.uml4spm.uml4spm.*;
import org.eclipse.emf.ecore.*;
import org.eclipse.uml2.uml.*;
.....

public class ActivityExecution extends Execution {

   //the Activity execution name (same as the activity in the model)
   public String name;
   // a reference towards the Activity definition in the model
   public SoftwareActivity activityType=null;
   //vectors to store runtime instances of the ActivityExecution
   public  List<ActivityEdgeInstance> activityEdgeInstances;
   public  List<ActivityNodeExecution> activityNodeExecInstances;
   //Hashtables to store the mapping between an Activity element and its
   //equivalent in runtime instances
   public  Hashtable<ActivityNode, ActivityNodeExecution> ActivityNodesMap;
   public  Hashtable<ActivityEdge, ActivityEdgeInstance> ActivityEdgesMap;
   //Parameter of the Sofwtare Activity
   public List<ActivityParameterNodeExecution> inputActivityParamNodeExecution;
   public List<ActivityParameterNodeExecution> outputActivityParamNodeExecution;


   //Constructor
   public ActivityExecution(SoftwareActivity activity){

          //give it the same name as the original activity
          name=activity.getName();

          //keep a link to its definition =>trace between activity definition
          //and its execution
          activityType=activity;

          //Variables initialisation - vectors to store runtime instances of the
          //ActivityExecution
          activityEdgeInstances=new Vector <ActivityEdgeInstance>();
          activityNodeExecInstances=new Vector <ActivityNodeExecution>();

          //Variables initialisation - Hashtables to store the mapping between
          //an Activity element and its equivalent in runtime instances
          ActivityNodesMap=new Hashtable<ActivityNode, ActivityNodeExecution>();
          ActivityEdgesMap=new Hashtable <ActivityEdge, ActivityEdgeInstance>();

          //Activity Parameter Nodes nitialization
          inputActivityParamNodeExecution=new
          Vector<ActivityParameterNodeExecution>();
          outputActivityParamNodeExecution=new
          Vector<ActivityParameterNodeExecution>();
   }

          //Constructor in case of an UML AD Diagram
          /*public ActivityExecution(Activity activity){

                       //give it the same name as the original activity
                       name=activity.getName();
                       //keep a link to its definition =>trace between activity
                       definition and its execution
                       activityType=activity;

                       //Variables initialisation - vectors to store runtime
                       instances of the ActivityExecution
                       activityEdgeInstances=new Vector
                       <ActivityEdgeInstance>();
                       activityNodeExecInstances=new Vector
                       <ActivityNodeExecution>();

                       //Variables initialisation - Hashtables to store the
                       //mapping between an Activity element and its equivalent
                       //in runtime instances
                       ActivityNodesMap=new Hashtable<ActivityNode,
                       ActivityNodeExecution>();
```

```
            ActivityEdgesMap=new Hashtable <ActivityEdge,
            ActivityEdgeInstance>();
        }*/


//a method that creates for each ActivityEdge its equivalent in runtime
public void CreateActivityEdgeInstance(ActivityEdge actEdge){

    // Create the run time equivalent
    ActivityEdgeInstance activityEdgeI=new ActivityEdgeInstance(actEdge,
    this);
    //add the runtime instance to the ActivityExecution edges table
    activityEdgeInstances.add((ActivityEdgeInstance)activityEdgeI);
    //Keep a trace between the activity definion and its equivalent at
    //runtime
    ActivityEdgesMap.put(actEdge, activityEdgeI);

}

//a method that creates for each Action, depending on its type, an
//ActionExecution instance
public void CreateActionExecutionInstance(Action action){

    //get the ActionExecution class type for this Action through the
    //HashTable we defined
    String runClassName=
    Configuration.MetaClassesMapping.get(action.eClass().getName());

    try{
        //load the ActionExecution class equivalent to the Action type
        //in the AD
        Class runclass=Class.forName(Configuration.classExecPath +
        runClassName);

        //find the class type of the Action given in the AD in order to
        //pass it to the apropriate ActionExecution Constructor
        Class umlDefClass=
        Class.forName(Configuration.classUMLDefPath+action.eClass().getN
        ame());

        System.out.println(" Action kind to instantiate is :"+
        umlDefClass.getName());


        //Initialize the params of the getConstructor method and the
        //newInstance method
        Class[] tab={umlDefClass, ActivityExecution.class};

        Object[] obj={action, this};

        //Create the Exectuable instance equivalent to the Action
        //defined in the AD
        ActionExecution actionExecInstance=
        (ActionExecution)(runclass.getConstructor(tab)).newInstance(obj)
        ;

        //add the runtime instance to the ActivityNodeExecInstance table

        activityNodeExecInstances.add((ActivityNodeExecution)actionExecI
        nstance);

        //Keep a trace between the action definion and its equivalent
        //runtime instance
        ActivityNodesMap.put((ActivityNode)action,
        (ActivityNodeExecution)actionExecInstance);

    }
    catch (Exception e){
        System.out.println("Linkage Failed while loading class!!!!");
        e.printStackTrace();
    }
}
```

```
//a method that creates for each control node, its equivalent runtime
//instance
public void CreateControlNodeExecutionInstance (ControlNode controlNode){

       //get the ControlNodeExecution class type for this ControlNode through
       //the HashTable we defined
       String runClassName=
       Configuration.MetaClassesMapping.get(controlNode.eClass().getName());

       try{
             //load the ControlNodeExecution class equivalent to the
             //ControlNode type in the AD
             Class runclass=Class.forName(Configuration.classExecPath +
             runClassName);

             //find the class type of the ControlNode given in the AD in
             //order to pass it to the appropriate ControlNodeExecution
             //Constructor
             Class umlDefClass=
             Class.forName(Configuration.classUMLDefPath+controlNode.eClass()
             .getName());

             //Initialize the params of the getConstructor method and the
             //newInstance method in two steps :

             //1- Define the Class types of parameteres
             Class[] tab={umlDefClass, ActivityExecution.class};

             //2- Define Parameters for creating the new instance.
             //They consist in the Control Node from the AD and the
             //ActivityExecution instance that will own the
             //ControlNodeExecution instance
             Object[] obj={controlNode, this};

             //Create the Exectuable instance equivalent to the Control Node
             //defined in the AD
             ControlNodeExecution controlNodeExecInstance=
             (ControlNodeExecution)(runclass.getConstructor(tab)).newInstance
             (obj);

             //add the runtime instance to the ActivityNodeExecInstance table

             activityNodeExecInstances.add((ActivityNodeExecution)controlNode
             ExecInstance);

             //Keep a trace between the Control definion and its equivalent
             //runtime instance
             ActivityNodesMap.put((ActivityNode)controlNode,
             (ActivityNodeExecution)controlNodeExecInstance);

       }
       catch (Exception e){
             System.out.println("Linkage Failed while loading class!!!!");
             e.printStackTrace();
       }

}

//a method that creates for each Object node, its equivalent runtime instance
public void CreateObjectNodeExecutionInstance (ObjectNode objectNode){

       //get the ControlNodeExecution class type for this ControlNode through
       //the HashTable we defined
       String runClassName=
       Configuration.MetaClassesMapping.get(objectNode.eClass().getName());

       if (runClassName.equalsIgnoreCase("ActivityParameterNodeExecution")){
             try{
                   //load the ObjectNodeExecution class equivalent to the
                   //ControlNode type in the AD
                   Class runclass=Class.forName(Configuration.classExecPath
                   + runClassName);

                   //find the class type of the ObjectNode given in the AD
                   //in order to pass it to the appropriate
                   //ObjectNodeExecution Constructor
```

```
                Class umlDefClass=
                Class.forName(Configuration.classUMLDefPath+objectNode.eC
                lass().getName());


                //Initialize the params of the getConstructor method and
                //the newInstance method in two steps :

                //1- Define the Class types of parameteres
                Class[] tab={umlDefClass, ActivityExecution.class};

                //2- Define Parameters for creating the new instance.
                //They consist in the Object Node from the AD and the
                //ActivityExecution instance that will own the
                //ObjectNodeExecution instance
                Object[] obj={objectNode, this};

                //Create the Exectuable instance equivalent to the Object
                //Node defined in the AD
                ActivityParameterNodeExecution
                ActivityParameterNodeInstance=
                (ActivityParameterNodeExecution)(runclass.getConstructor(
                tab)).newInstance(obj);

                //add the runtime instance to the
                //ActivityNodeExecInstance table

                activityNodeExecInstances.add((ActivityNodeExecution)Acti
                vityParameterNodeInstance);

                //Keep a trace between the Object Node definion and its
                //equivalent runtime instance
                ActivityNodesMap.put((ActivityNode)objectNode,
                (ActivityNodeExecution)ActivityParameterNodeInstance);

                if (objectNode.getIncomings().isEmpty()){


                    this.inputActivityParamNodeExecution.add(ActivityPa
                    rameterNodeInstance);

                }
                if (objectNode.getOutgoings().isEmpty()){


                    this.outputActivityParamNodeExecution.add(ActivityP
                arameterNodeInstance);

                }

            }

            catch (Exception e){
                System.out.println("Linkage Failed while loading
                class!!!!");
                e.printStackTrace();
            }
        }
    }

//A method that links together AcitvityEdgeInstances and ActivityExecutionNodes
//(actions, control nodes)
public void linkEdgeInstancesToActivityExecNodes(){
        System.out.println("la taille de activityEdgeInstances est de :
        "+activityEdgeInstances.size());
        Iterator iterator=activityEdgeInstances.iterator();
        while (iterator.hasNext()){
            ActivityEdgeInstance actEdgeI=
            (ActivityEdgeInstance) iterator.next();
            //1-step: Set the Source and Target (Activity Execution Nodes
            //Instances) Propery of the Edge Instance

            actEdgeI.source=(ActivityNodeExecution)ActivityNodesMap.get(actE
            dgeI.edge.getSource());

            actEdgeI.target=(ActivityNodeExecution)ActivityNodesMap.get(actE
            dgeI.edge.getTarget());
            //2-step: Set this Edge Instance as IncomingEdge of the
            //ActivityExecNode's Target property and as OutgoingEdge of the
            //ActivityExecNode's Source property

               ((ActivityNodeExecution)actEdgeI.source).addOutgoingEdge(actE
            dgeI);

                   ((ActivityNodeExecution)actEdgeI.target).addIncomingEdge(
            actEdgeI);
        }

    }
    //Defines which Executable Class to instantiate depending on its type
    public void activityElementSort(Collection activityElements){

         Iterator it = activityElements.iterator();
         while(it.hasNext()) {

           EObject element =(EObject) it.next();

           //get the super class of the element, if it is either an ActivityEdge
           //or an ActivityNode (Actions or Control Nodes)
           EList superClass=element.eClass().getEAllSuperTypes();

           if (superClass!=null){
                 if (hasSuperType(superClass,"Action")){

                       //Call a method that will create an ActionExecution
                       //Instance depending on the Action Type
                       //(CallOperationAction, OpaqueAction,.etc)
                       CreateActionExecutionInstance((Action)element);
                       System.out.println ("element added : " +
                       element.eClass().getName());
                       }
                 if (hasSuperType(superClass,"ActivityEdge")){
                       //Call a method that will create ActivityEdge Instances
                       CreateActivityEdgeInstance((ActivityEdge)element);
                       System.out.println ("element added : " +
                       element.eClass().getName());
                 }
                 if (hasSuperType(superClass,"ControlNode")){
                       //Call a method that will create a ControlNodeExecution
                       //Instance depending on the Control node Type (Initial
                       //Node, Fork Node, Merge Node,.etc)
                       CreateControlNodeExecutionInstance((ControlNode)element);
                       System.out.println ("element added : " +
                       element.eClass().getName());
                 }
                 if (hasSuperType(superClass,"ObjectNode")){
                       //Call a method that will create a ObjectNodeExecution
                       //Instance depending on the Control node Type (Initial
                       //Node, Fork Node, Merge Node,.etc)
                       CreateObjectNodeExecutionInstance((ObjectNode)element);
                       System.out.println ("element added : " +
                       element.eClass().getName());
                 }
           }
           else
                 System.out.println("Unkwon UML super class for this element" +
                 element.eClass().getName());
         }
              System.out.println("la taille de la collection
           ActivityEdgesInstance est de :" + activityEdgeInstances.size());

      }


    public  void intialize(){
    //Initialize the mapping between class definitions and their runtime classes.
    //If you want to extend the engine, you have just to add new concepts in
    //configuration and their mappings
           Configuration.configurationMap();

    //Sort Activity elements by Actions, Control Nodes, Object Nodes and Edges
    //and then, create them
           activityElementSort(activityType.allOwnedElements());
```

```
          //Link the Activity Edge Instances with Activity Node Execution
          linkEdgeInstancesToActivityExecNodes();


   }
   //Load process WorkProducts on activity input pins without incoming edges
   public void loadWorkProductToActivityObjectNodes(InputPinExecution inPinExec,
   String workProductType){
          ObjectToken objToken=new ObjectToken();

          objToken.setReferencedWorkProduct(ProcessModelExecution.workProductLis
          t.get(workProductType));
          if (objToken.getReferencedWorkProduct()!=null){
                 inPinExec.offeredTokens.add(objToken);
                 inPinExec.offering=true;
                 }
          else{
                 System.out.println("ERROR: no such workProduct Type :"+
                 workProductType+ " available to be loaded in ObjectNode :"
                 +inPinExec.name);
          }


   }
   public boolean execute(){
           System.out.println("Starting the execution of :"+this.name);
          //Find all Inputpins without incoming edges
          Collection <InputPinExecution> inputPinsWithoutIncomingEdges=
                 new Vector<InputPinExecution>();
          Collection <InputPin> inputPins=
          findElementByType(ActivityNodesMap.keySet(), "InputPin");
          if (!(inputPins.isEmpty())){
                 Iterator it=inputPins.iterator();
                 //browse all input pins
              while (it.hasNext()){
                 InputPin inputPin=(InputPin)it.next();
                 //retain only inputpins without incoming edges
                 if (inputPin.getIncomings().size()==0){
                        InputPinExecution
                        inPinExec=(InputPinExecution)ActivityNodesMap.get(inputPi
                        n);
                        String workProductType=inputPin.getType().getName();

                        inputPinsWithoutIncomingEdges.add(inPinExec);

                        loadWorkProductToActivityObjectNodes(inPinExec,workProduc
                 tType );
                 }

              }

          }
          //Find all Intial Nodes and Fire them
          Collection <InitialNode> intialNodes=
          findElementByType(ActivityNodesMap.keySet(), "InitialNode");
          if (!(intialNodes.isEmpty())){
                 Iterator iter=intialNodes.iterator();
                 //browse all initial nodes
              while (iter.hasNext()){
                 InitialNodeExecution
                 iniNodeExec=(InitialNodeExecution)ActivityNodesMap.get((InitialN
                 ode)iter.next());
                 //fire them
                 iniNodeExec.fire();
              }

          }
          return true;
   }

}
```

ActivityEdgeInstance class

```
package ExecActivity;


import java.util.Vector;

import move.lip6.uml4spm.uml4spm.UML4SPMPackage;
import move.lip6.uml4spm.uml4spm.WorkProduct;

import org.eclipse.uml2.uml.ActivityEdge;
.......
.......

public  class ActivityEdgeInstance {
   //name
   String name;
   //the ActivityExecution owning this edge
   public ActivityExecution context=null;
   //the edge for which this instance is its runtime instance
   public ActivityEdge edge=null;
   //The source Acitivity Node Execution of this edge
   ActivityNodeExecution source=null;
   //The target Acitivity Node Execution of this edge
   ActivityNodeExecution target=null;

   //Constructor
   public ActivityEdgeInstance(ActivityEdge actEdge, ActivityExecution context){
             edge=actEdge;
             name=actEdge.getName();
             this.context=context;
      }


   public String getName(){
         return this.name;
   }
   public ActivityEdge getActivityEdge(){
         return this.edge;
   }
   public void setActivityEdge(ActivityEdge edge){
         this.edge=edge;
   }
   public ActivityNodeExecution getSource(){
         return this.source;
   }
   public ActivityNodeExecution getTarget(){
         return this.target;
         }

   //Send (forward from the source node) an offer
   public void sendOffer(){
         //Before sending an offer, check whether the Edge has a Guard
         if (hasGuard())
               //if the Edge has a guard, then evaluate it
               if (evaluateGuard())
               //if guard evaluation returns true, then forward the offer
                     this.getTarget().receiveOffer();
               else
                     System.out.println("Guard   Expression   Evaluation   of
                     Activty Edge :"+this.name+ " failed!!. Can't send offer
                     to the Activity Node :" + this.target.name);
         else
               this.getTarget().receiveOffer();
   };

//returns offered tokens from the source(called from the target activity node)
   public Vector <Token> takeTokens(){

         return (this.getSource().takeOfferedTokens());
   };
   public int countOfferedTokens(){

         int nbTokens=0;
         //...
         return nbTokens;
   };
   public boolean sourceHasOffer(){

      return source.hasOffer();
   };

   public String getGuardString(){
      return this.getActivityEdge().getGuard().stringValue();

   }
   //check if the edge has a guard
   public boolean hasGuard(){
      return (this.getActivityEdge().getGuard()!=null);
   }
   //evaluate the Activity Edge Guard Expression
   public boolean evaluateGuard(){
      boolean resultGuardEvaluation=false;
      if (this.hasGuard()){
            //get the Guard value (String) to be evaluated
             String toevaluate=this.getGuardString().trim();
            //extract the name of the concerned WorkProduct to evaluate
            String workproduct=toevaluate.substring(0, toevaluate.indexOf("."));
            //extract the name of the workproduct attribute to evaluate
            String property=
            toevaluate.substring(toevaluate.indexOf(".")+1,
          toevaluate.indexOf("="));
            //extract the value of the attribute from the guard expression
           String valueProperty=
          toevaluate.substring(toevaluate.indexOf("=")+1, toevaluate.length());
            //get the a workproduct from the Process Model list of workproducts
            WorkProduct wproduct=
             ProcessModelExecution.workProductList.get(workproduct);
            //extract the structural feature (attribute) to evaluate
            EStructuralFeature sFeature=
          UML4SPMPackage.eINSTANCE.getWorkProduct().getEStructuralFeature(proper
          ty);
            //check that the value given in the guard expression equals the
            //workproduct's attribute value
            if (wproduct.eGet(sFeature).toString().equals(valueProperty)){
                  System.out.println("Guard expression :"+ toevaluate+ ", of the
                  Object Flow :"+ this.name +" evaluated at TRUE ");
                  resultGuardEvaluation=true;
            }
            else
                  System.out.println("Guard expression :"+ toevaluate+ ", of the
                  Object Flow :"+ this.name +" evaluated at FALSE ");

      }
      else
             System.out.println("the Activity Edge does not have a Guard");
    return resultGuardEvaluation;
   }

}
```

ActivityNodeExecution class

```
package ExecActivity;


import org.eclipse.uml2.uml.ActivityNode;
.......

public abstract class ActivityNodeExecution {

   public String name;
   public ActivityExecution activityExecContext=null;
   public org.eclipse.uml2.uml.ActivityNode activityNode=null;
   public boolean offering=false;
   public boolean terminated=false;

   public List<Token> offeredTokens = new Vector<Token>();
   public List<ActivityEdgeInstance> outgoingEdges = null;
   public List<ActivityEdgeInstance> incomingEdges = null;

   public ActivityNodeExecution(ActivityNode actNode, ActivityExecution
context){
          this.incomingEdges=new Vector<ActivityEdgeInstance>();
          this.outgoingEdges=new Vector<ActivityEdgeInstance>();
          //the name of the activity Node
          this.name=actNode.getName();
          //the context (activity) of the activity node
          this.activityExecContext=context;

   }

   //added Method to check wether the node has incoming edges or not
   public boolean hasIncomingEdges(){
          return (!(incomingEdges.isEmpty()));
   }
   public void receiveOffer(){
          //Call the isReady method to check is the node is ready to execute
          if (this.isReady()){
                this.fire();
          }
   };

   public boolean isReady(){
          boolean isReady=false;
          //For Acivity Nodes having incoming edges
          if (hasIncomingEdges()){
                  Iterator it = incomingEdges.iterator();
                  while(it.hasNext()) {
                       ActivityEdgeInstance aEI=(ActivityEdgeInstance)it.next();
                       //check if the source Activity Node has an offer
                       isReady=aEI.sourceHasOffer();
                  }
          }
          else {
                //noeud whithout incoming edges or initial node
                isReady=true;
          }
          if (!isReady)
                System.out.println("Activity Node :"+this.name+ ", not
                ready!!!!!");
          else
                System.out.println("Activity Node :"+this.name+ ", is
                ready!!!!!");
          return isReady;
   };


   public Vector <Token> takeOfferedTokens(){
          //creattion of a new vector of Tokens that will be returned
          //by the method before clearing the original offeredTokens Vector
          Vector<Token> offeredTokensToTarget=new Vector<Token>();
           if (!(this.offeredTokens.isEmpty())){
                       offeredTokensToTarget.addAll(this.offeredTokens);
                       // The node is no longer offering Tokens
                       this.offering=false;
                //Clearing the tokens offered by this node to the target node
                        this.offeredTokens.clear();
                        //retun the offeredTokens to the target node
         }
         else
                 System.out.println("The Node has no Tokens to offers!!!!");

         return offeredTokensToTarget;
    };
    public abstract void fire();

    public ActivityExecution getActivityExecution(){

            return activityExecContext;
    };

    public int countOfferedTokens(){
            int nbOfferedtokens=0;
            //...
            return nbOfferedtokens;

    };

    public void sendOffer (){
            if (!(this.outgoingEdges.isEmpty())) {
                    Iterator<ActivityEdgeInstance> it=this.outgoingEdges.iterator();
                    while (it.hasNext()){
                            ActivityEdgeInstance aeInstance=it.next();
                            aeInstance.sendOffer();
                    }
            }
    };

    public void addIncomingEdge (ActivityEdgeInstance edge){
            incomingEdges.add(edge);
    };
    public void addOutgoingEdge (ActivityEdgeInstance edge){
            outgoingEdges.add(edge);

    };
    public boolean hasOffer(){
            return offering;
    };

    public void terminate(){};

    //Not implemented in the context of Software Process execution
    public Object getExecutionContext (){
            Object context=null;
            //.....
            return context;

    };

    //Not implemented in the context of Software Process execution
    public Location getExecutionLocation(){

            Location location=null;
            //....
            return location;
    };

}
```

MergeNodeExecution class

```
public class MergeNodeExecution extends ControlNodeExecution {
   ........

   public MergeNodeExecution(MergeNode mergeNode, ActivityExecution context){
          super(mergeNode,context);
          this.mergeNodeLink=mergeNode;
   }

   //an offer is made to an ActivityNodeExecution by
   //calling its receiveOffer()
   //we redefined this operation in the context of the MergeNode
   //when the Merge Node receives an offer it simply forward it
   //to the target ActivityNodeExecution

   public void receiveOffer() {

          //when the MergeNodeExecution receives an offer
   //it does not need to call isReady()-> always true
   //fire the behavior of the MergeNode

          this.fire();
 }

   //simply forward the offer to the outgoing edge
   //(transitively), to the following ActivityNodeExecution
   //the ActivityNodeExecution will then check that all its
   //input pins/control flows are ready and then will fire them

 public void fire(){

   //Check that the MergeNode has outgoing edges
   if (!(this.outgoingEdges.isEmpty())){

          //Check that the MergeNode does not have more than one outgoing edge
          if (this.outgoingEdges.size()<2){

                //send an offer on its outgoing edges.
                //In the case of MergeNode, there is only one outgoing edge
                this.outgoingEdges.get(0).sendOffer();

                }
                else
                  System.out.println("ERROR: Merge Node " + this.name+
                       ", must not have more than one outgoing edge");
          }
          else
                System.out.println("Merge Node :" + this.name +
                       ", has no outgoing edge");

   }

   //an operation that takes the ActivityNodeExecution offered tokens
   //we redefined this operation for the MergeNodeExecution
   //now, it simply forward the request to the target ActivityNodeExecution

 public Vector<Token> takeOfferedTokens() {
          //forward the request to the source ActivityNodeExecution.
          //tokens will be directly taken form source ActivityNodeExecution
          //to the target ActivityNodeExecution

   return (this.incomingEdges.get(0).takeTokens());

   }
   //an operation that checks if the ActivityNodeExecution is making an offer
   //redefined in the context of the MergeNode
   //now it simply forward the offer to the source ActivityNodeExecution

 public boolean hasOffer() {

   //forward the request to the source ActivityNodeExecution
   return this.incomingEdges.get(0).sourceHasOffer();
   }
   ......
   }
```

DecisionNodeExecution's *fire()* operation

```
public void fire(){

       //check that the Decision node has one and only one incoming edge
        if (!(this.incomingEdges.isEmpty())){
             if (this.incomingEdges.size()<2){
                    Iterator it=outgoingEdges.iterator();
                    boolean targetHasGuard=false;

                    //check if Decision Node outgoing edges have guards
                    while (it.hasNext()){

                    targetHasGuard=((ActivityEdgeInstance)it.next()).hasGuard();
                    }

                    //if they have guards, send an offer to the outgoing edges
                    //the evaluation of the guards is carried out by the
                    //ActivityEdgeInstance
                    if (targetHasGuard){
                           this.sendOffer();
                    }

                    //if no guards are specficied on outgoing edges
                    //ask the agent to choose between the possible outgoing edges
                    else{
                           System.out.println("Traget Activity Edges do not have
                           Guards,"+ "please choose one between activity edge
                           targets");

                           int i=1;

                           Iterator iter=outgoingEdges.iterator();

                           while (iter.hasNext()){
                                 System.out.println("enter :"+i+ "for selecting:"  +
                                 ((ActivityEdgeInstance)iter.next()).name);

                                 i++;
                           }

                           //read the agent choice
                           Scanner KeyBoard=new Scanner(System.in);

                           int choice=KeyBoard.nextInt();

                    //send an offer on the outgoing edge chosen by the agent
                           this.outgoingEdges.get(choice-1).sendOffer();

                    }

             }
             else
                    System.out.println("ERROR!: a Decision Node must not have more
                    thant one incoming edge!!!");
        }
        else
             System.out.println("This Decision Node :" + this.name+" has no
             incoming edge!!!");

             }
```

InputPinExecution's *fire()* operation.

```
//InputPinexecution is fired by
//the Action Execution owning it

public void fire(){

        //check that node has incoming edges

        if (!(this.incomingEdges.isEmpty())) {

                Iterator<ActivityEdgeInstance> it=incomingEdges.iterator();

                while (it.hasNext()){

                        ActivityEdgeInstance aeInstance=it.next();

                        //take tokens offerd by source ActivityNodeExecution
                        //by passing by the intermadiate ActivityEdgeInstance

                        this.offeredTokens.addAll(aeInstance.takeTokens());

                }
        }
}
```

ActivityParameterNodeExecution Class

```
package ExecActivity;

import java.util.Iterator;

import org.eclipse.uml2.uml.ActivityParameterNode;

public class ActivityParameterNodeExecution extends ObjectNodeExecution{


        //keep a link with the pin definition
        public ActivityParameterNode activityParameterDefinitionLink;


        public ActivityParameterNodeExecution(ActivityParameterNode
        aParamNode, ActivityExecution activityContex){
                super(aParamNode,activityContex);
                this.name=aParamNode.getName();

                //activity context of the action containing that pin
                this.activityExecContext=activityContex;
                //link with the original pin
                this.activityParameterDefinitionLink=aParamNode;

        }


        public void fire(){
                //Now, the Activity Parameter is ready
                this.offering=true;
                //In case of APN in
                if (this.incomingEdges.isEmpty()) {
                        if (!(this.outgoingEdges.isEmpty())){
                                Iterator<ActivityEdgeInstance> it=
                                outgoingEdges.iterator();
                        while (it.hasNext()){
                                ActivityEdgeInstance aeInstance=it.next();
                                aeInstance.sendOffer();
                        }
                        }
                        else
                                System.out.println("ERROR in Model : Activity
                                Parameter Node :"+ this.name+", is isolated (i.e.,)
                                neither incoming nor outgoing edges");
                }
                System.out.println("La Taille de liste de APN de :" +
                this.activityExecContext.name+ "  est de :"
                +this.activityExecContext.inputActivityParamNodeExecution.size()
                );

                //In case of APN out
                if (this.outgoingEdges.isEmpty()) {
                        if (!(this.incomingEdges.isEmpty())){
                                Iterator<ActivityEdgeInstance> it=
                                incomingEdges.iterator();
                        while (it.hasNext()){
                                ActivityEdgeInstance aeInstance=it.next();
                                this.offeredTokens.addAll(aeInstance.takeTokens());
                        }
                        }
                        else
                                System.out.println("ERROR in Model : Activity
                                Parameter Node :"+ this.name+", is isolated (i.e.,)
                                neither incoming nor outgoing edges");
                }
                //....
        }
}
```

CallBehaviorActionExecution's *doAction()* operation

```
//The CallBehaviorActionExecution's main behavior
   public void doAction(){


          //Check IN / OUT parameters (types + nbr of arguments) of the //call
          action with those of the called behavior

          boolean conforms=checkCallParametersConformity();

          if (conforms){

                //initialize the Activity Parameter node of the called
                //behavior and fire them.
                initializeCalledBehaviorActPNodes();

                       //check if the call is asynchronous
                       if (isSynchronousCall(this)){

                       //get the results of the call and put them in //outputpin
                       execution instances accordingly
                             getCallResult();

                             //Fire the execution of all output pins
                             fireOutputPins();

                       }

                       // consume the input pins of the action
                       consumeActionInputPins();

                       //prepare to offer a control flow token on //outcoming
                       edges
                       putControlToken();

                       //send offer on outgoing edges
                       sendOffer();

          }
          else
                System.out.println("Call Parameters do not match Acitivity
                Parameter Nodes of the called behavior (in nbr or in
                types)!!!");

          }
```

OpaqueActionExecution Class

```
package ExecActivity;

import org.eclipse.uml2.uml.*;

public class OpaqueActionExecution extends ActionExecution{

        //keep trace of the OpaqueAction definition in the input model
        public org.eclipse.uml2.uml.OpaqueAction oAction=null;

        //Constructor
public OpaqueActionExecution(OpaqueAction opAction, ActivityExecution context) {
                super(opAction,context);

                //keep trace of the OpaqueAction definition in the input model
                this.oAction=(OpaqueAction)opAction;
        }

public void putObjectTokensToOuputPins(OutputPinExecution outPinExec, String
workProductType){
                ObjectToken objToken=new ObjectToken();
                objToken.setReferencedWorkProduct(ProcessModelExecution.workProductLis
                t.get(workProductType));

                if (objToken.getReferencedWorkProduct()!=null){
                        outPinExec.offeredTokens.add(objToken);
                        outPinExec.offering=true;
                        }
                else{
                        System.out.println("ERROR: no such workProduct Type :"+
                        workProductType+ " available to be loaded in ObjectNode :"
                        +outPinExec.name);
                }
        }
        public void doAction() {

                try {
                        //the run class is given further in this appendix
                        RunClass.createClass(oAction);
                        //executeBody must basically returns the set of Object Token
                        //referencing existing or newly created workproducts
                        boolean resultExec=RunClass.executeBody(oAction);
                        if (resultExec){
                                if (!(this.actionOutPutPinExecInstances.isEmpty())){
                                        Iterator it=
                                        actionOutPutPinExecInstances.iterator();
                                        while (it.hasNext()){
                                        OutputPinExecution ouputPinExec=
                                        (OutputPinExecution)it.next();
                                        //Get the type of the output pin
                                        String outputType=
                                        ouputPinExec.pinDefinitionLink.getType().getName();
                                        //create and put object tokens on action outputs
                                        //Once the GUI of the opaque action realized,
                                        //Object Tokens will be automatically generated
                                        from user interactions with the GUI : create a new
                                        //Worproduct, modify an existing one, etc.
                                        //for prototyping purposes, we create the object
                                        //token automatically according to their types
                                        putObjectTokensToOuputPins(ouputPinExec,
                                        outputType);
                                        //fire output pins
                                        fireOutputPins();
                                        }
                                }
                                putControlToken();
                                this.sendOffer();
                        }
                }
                catch (Exception e){
                                 e.printStackTrace();
                        }
                        }
}
```

RunClass Class (used by OpaqueActionExecution)

```
package ExecActivity;

import java.util.*;
.........

.........
public class RunClass {

       public static void createClass(OpaqueAction oAct){
              List list=oAct.getBodies();
       if (list.size()!=0){

        String body=(String)list.get(0);
       try {
           // Prepare the signature of the class.
          //Its name will the same as the opaque Action name
             String [] classDef={"package ExecActivity;", "public class "
             +oAct.getName()+"{", "public void ExecuteBody(){", body, "}}"} ;
         //Create the file
             PrintWriter out = new PrintWriter(new BufferedWriter(new
             FileWriter("src/ExecActivity/"+oAct.getName()+".java")));

          // fill the content of the class with the Opaque Action body
             for (int i=0; i<classDef.length; i=i+1){
                    out.println(classDef[i]);
           }
        out.close();
        //compile the classe definition
        compileClass("src/ExecActivity/".concat(oAct.getName().concat(".java")));
       }
       catch (IOException e){

          e.printStackTrace();
       }
              }
       }
       public static void compileClass(String className){
              String argument[]={"javac", "-d", "bin",className};

              try {
              Runtime rt=Runtime.getRuntime();
              Process process=rt.exec(argument);
              process.waitFor();
              System.out.println("Compilation succeeded!!!");
              }
              catch (IOException ioe) {
                     System.out.println("Problem encountred while compiling
                     "+className);
              } catch (InterruptedException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              }
       }
       public static boolean executeBody(OpaqueAction oAct){
              boolean execute=true;
              try {
              java.lang.Class cl =
              java.lang.Class.forName("ExecActivity."+oAct.getName());

              java.lang.reflect.Method m;

              m = cl.getMethod("ExecuteBody", (java.lang.Class [])null);

              m.invoke(cl.newInstance(), (Object[])null);

              } catch(Exception e) {
                     e.printStackTrace();
                     execute=false;
              }
              return execute;
       }

                                          }
```

Execution Traces of the Software Process Example using the UML4SPM Execution Model Appraoch

The process engine starts by loading the process model and by instantiating, for each element in the process model, its equivalent execution class. We can see traces of the creation of these execution classes. Then it looks for the *initial* software activity (Inception in this case) and starts its execution.

While the process is executing we can see traces related to the fact that some activity node executions are ready to execute, that some CallBehaviorAction call parameters are checked with the ActivityParamaterNodes of the called software activity, etc. We can also notice the guard evaluation results of the *Decision_To_SendFailMessage* and *Decision_To_SendSuccMessage* object flows.

For the execution of OpaqueActions, we can see traces like "compilation succeeded!!!" followed by the text "I am in OpaqueActionName". This text is part of a Java instruction we inserted in the Opaque Action's body property at modeling time. This trace proves that the Java instruction we specified within the body property has been executed at runtime and without interrupting the process execution.

Finally, the *Construction* phase is called asynchronously before terminating the *Inception Phase* execution. In case of a process model with interaction points, the process engine asks the agent for entries before continuing its execution.

The execution time of the entire process model takes less than three seconds.

```
Program Start
SoftwareProcessExample
The Software Activity name is :Inception
element added : InitialNode
Action kind to instantiate is :org.eclipse.uml2.uml.CallBehaviorAction
Checking if the Action :ElaborateAnalysisModel has Input Pins, if so, create them
The action has an input named :requirementDocument
Checking if the Action :ElaborateAnalysisModel has Output Pins, if so, create them
The action has an Output named :umlAnalysisModel
element added : CallBehaviorAction
 Action kind to instantiate is :org.eclipse.uml2.uml.CallBehaviorAction
Checking if the Action :ValidateAnalysisModel has Input Pins, if so, create them
The action has an input named :umlAnalysisModel
Checking if the Action :ValidateAnalysisModel has Output Pins, if so, create them
The action has an Output named :validationReport
element added : CallBehaviorAction
element added : DecisionNode
 Action kind to instantiate is :org.eclipse.uml2.uml.OpaqueAction
Checking if the Action :SendFailMessage has Input Pins, if so, create them
The action has an input named :validationReport_Fail
Checking if the Action :SendFailMessage has Output Pins, if so, create them
element added : OpaqueAction
 Action kind to instantiate is :org.eclipse.uml2.uml.OpaqueAction
Checking if the Action :SendSuccMessage has Input Pins, if so, create them
The action has an input named :validationReport_Succ
Checking if the Action :SendSuccMessage has Output Pins, if so, create them
element added : OpaqueAction
 Action kind to instantiate is :org.eclipse.uml2.uml.CallBehaviorAction
Checking if the Action :ConstructionPhase has Input Pins, if so, create them
Checking if the Action :ConstructionPhase has Output Pins, if so, create them
element added : CallBehaviorAction
element added : MergeNode
element added : ActivityFinalNode
element added : ControlFlow
element added : ObjectFlow
element added : ObjectFlow
element added : ObjectFlow
element added : ObjectFlow
element added : ControlFlow
element added : ControlFlow
element added : ControlFlow
element added : ControlFlow
element added : OutputPin
element added : InputPin
element added : OutputPin
element added : InputPin
element added : InputPin
element added : InputPin
la taille de la collection ActivityEdgesInstance est de :9
la taille de activityEdgeInstances est de : 9
The Software Activity name is :ElaborateAnalysisModel
element added : ActivityParameterNode
 Action kind to instantiate is :org.eclipse.uml2.uml.OpaqueAction
Checking if the Action :ElaborateUMLAnalysisModel has Input Pins, if so, create
them
The action has an input named :requirementDocument_ElaborateAN
Checking if the Action :ElaborateUMLAnalysisModel has Output Pins, if so, create
them
The action has an Output named :umlAnalysisModel_ElaborateAN
element added : OpaqueAction
element added : ActivityParameterNode
element added : ObjectFlow
element added : ObjectFlow
element added : OutputPin
element added : InputPin
la taille de la collection ActivityEdgesInstance est de :2
la taille de activityEdgeInstances est de : 2
The Software Activity name is :Construction
la taille de la collection ActivityEdgesInstance est de :0
la taille de activityEdgeInstances est de : 0
The Software Activity name is :ValidateAnalysisModel
 Action kind to instantiate is :org.eclipse.uml2.uml.OpaqueAction
Checking if the Action :Check_and_EditValidationReport has Input Pins, if so,
create them
The action has an input named :UML_AnalysisModel_AN
Checking if the Action :Check_and_EditValidationReport has Output Pins, if so,
create them
The action has an Output named :validationReport_AN
element added : OpaqueAction
element added : ActivityParameterNode
element added : ActivityParameterNode
element added : ObjectFlow
element added : ObjectFlow
element added : OutputPin
element added : InputPin
la taille de la collection ActivityEdgesInstance est de :2
la taille de activityEdgeInstances est de : 2
Starting the execution of :Inception
Initial Node, Start of Software Activity :Inception fired
------------> Activity Node :ElaborateAnalysisModel, is ready!!!!!
------------> Activity Node :requirementDocument, is ready!!!!!
called actvitity is ElaborateAnalysisModel
Checking that nbr of the call action's inputs = the called activity parameters (in)
:true
the type of the action Pin is :RequirementDocument
the type of the activity parameter node is :RequirementDocument
Cheking that types of call action's inputs = types of the called activity's
parameter :true
Checking that nbr of the call action's outputs = the called activity parameters
(out):true
the type of the action Pin is :UMLAnalysisModel
the type of the activity parameter node is :UMLAnalysisModel
Cheking that types of call action's inputs = types of the called activity's
parameter :true
The Overall result of Checking if action call parameters (in/out) = the called
activity parameters (in/out) :true
------------> Activity Node :ElaborateUMLAnalysisModel, is ready!!!!!
------------> Activity Node :requirementDocument_ElaborateAN, is ready!!!!!
this Activity Node has no incoming edge -->:ElaborateUMLAnalysisModel
le nom de la classe est :ElaborateUMLAnalysisModel.java
Compilation succeeded!!!
I am in ElaborateUMLAnalysisModel Action
------------> Activity Node :umlAnalysisModel_APN, is ready!!!!!
La Taille de liste de APN de :ElaborateAnalysisModel  est de :1
La Taille de liste de APN de :ElaborateAnalysisModel  est de :1
```

```
------------> Activity Node :ValidateAnalysisModel, is ready!!!!!
------------> Activity Node :umlAnalysisModel, is ready!!!!!
this Activity Node has no incoming edge -->:ValidateAnalysisModel
called actvitity is ValidateAnalysisModel
Checking that nbr of the call action's inputs = the called activity parameters (in)
:true
the type of the action Pin is :UMLAnalysisModel
the type of the activity parameter node is :UMLAnalysisModel
Cheking that types of call action's inputs = types of the called activity's
parameter :true
Checking that nbr of the call action's outputs = the called activity parameters
(out):true
the type of the action Pin is :ValidationReport
the type of the activity parameter node is :ValidationReport
Cheking that types of call action's inputs = types of the called activity's
parameter :true
The Overall result of Checking if action call parameters (in/out) = the called
activity parameters (in/out) :true
------------> Activity Node :Check_and_EditValidationReport, is ready!!!!!
------------> Activity Node :UML_AnalysisModel_AN, is ready!!!!!
this Activity Node has no incoming edge -->:Check_and_EditValidationReport
le nom de la classe est :Check_and_EditValidationReport.java
Compilation succeeded!!!
I am in Check_and_EditValidationReport Action
------------> Activity Node :validationReport_APN, is ready!!!!!
La Taille de liste de APN de :ValidateAnalysisModel  est de :1
La Taille de liste de APN de :ValidateAnalysisModel  est de :1
------------> Activity Node :validationDecision, is ready!!!!!
Guard expression :ValidationReport.state=failed, of the Object Flow
:From_Decision_To_SendFailMessage evaluated at FALSE
Guard Expression Evaluation of Activty Edge :From_Decision_To_SendFailMessage
failed!!. Can't send offer to the Activity Node :validationReport_Fail
Guard expression :ValidationReport.state=validated, of the Object Flow
:From_Decision_To_SendSuccMessage evaluated at TRUE
------------> Activity Node :SendSuccMessage, is ready!!!!!
------------> Activity Node :validationReport_Succ, is ready!!!!!
this Activity Node has no incoming edge -->:SendSuccMessage
le nom de la classe est :SendSuccMessage.java
Compilation succeeded!!!
I am in SendSuccMessage Action
------------> Activity Node :ConstructionPhase, is ready!!!!!
called actvitity is ConstructionPhase
Checking that nbr of the call action's inputs = the called activity parameters (in)
:true
Cheking that types of call action's inputs = types of the called activity's
parameter :true
The Overall result of Checking if action call parameters (in/out) = the called
activity parameters (in/out) :true
Asynchronous Call to : ConstructionPhase
------------> Activity Node :Final, is ready!!!!!
Activity Final Node : Activity -> Inception  terminated
Execution End
```