

Achieving Process modeling and Execution through the Combination of Aspect and Model-Driven Engineering Approaches¹

Reda Bendraou^a, Jean-Marc Jezéquel^{b,c}, and Franck Fleurey^d

^a University Pierre & Marie Curie
4, Place Jussieu, Paris F-75005, France
{firstname.lastname@lip6.fr}

^b INRIA-Rennes Bretagne Atlantique, Campus de Beaulieu
F-35042 Rennes Cedex, France
{firstname.lastname@inria.fr}

^c IRISA, Université Rennes 1
Campus de Beaulieu

F-35042 Rennes Cedex, France
^d SINTEF, Oslo Franck.Fleurey@Sintef.no

Abstract. One major advantage of executable software process models is that once defined, they can be simulated, checked and validated in short incremental and iterative cycles. This also makes them a powerful asset for important process improvement decisions such as resource allocation, deadlock identification and process management. In this paper, we propose a framework that combines Aspect and Model-Driven Engineering approaches in order to ensure process modeling, simulation and execution. This framework is based upon UML4SPM, a UML2.0-based language for Software Process Modeling and Kermet, an executable metaprogramming language.

Keywords: Executable models, process modeling and execution, UML.

1 Introduction

Complementary to the use of traditional Verification and Validation (V&V) based approaches, it has been widely recognized that the quality of the software development process also has a direct impact on the quality of the software. By capturing team's best practices, task ordering, flows of artifacts, agent coordination and communications, process models become nowadays an important asset to ensure repeatability and quality in building software.

¹ This work is partially supported by the French ANR Project Galaxy (ref. ANR-09-SEGI-005) and the MOPCOM-I project from the Competitiveness Cluster of Brittany.

Recently, driven by the pressure for more accurate results and shorter time-to-market, a demand for executable process models emerged. Executable process models are process models that can be used not only for documenting processes and methods but also for the support of their execution. Indeed, executable process models can be used to coordinate between agents, to enforce artefacts routing between process's steps, to ensure rules and constraints integrity and process deadlines. They can also be of an effective aid since they can be used for simulation and testing. Simulation results can be used as a basis for important improvement decisions such as resource allocation, deadlock identification, estimation of the project duration and many other aspects that have a direct impact on the process and thus on the quality of the delivered software.

During the last two decades, the need for executable Software Process Modeling Languages (SPML) has been widely recognized. Osterweil opened the way with his seminal work "*Software Processes are Software Too*" (Osterweil 1987). He introduced the notion of *Process Programming*, which consisted in representing software processes in terms of computer-readable programs. The main goal behind this was to ensure agent coordination and the automation of process's repetitive and non-interactive tasks through the execution of *process programs*. The process programming trend stimulated many research works and had as an impact, the emergence of a multitude of SPMLs. These SPMLs were based on some well-known programming languages (e.g., Ada, LISP) or formal formalisms such as Petri Nets and put a strong emphasis on the executability aspect.

One of the lessons learned from these first-generation languages is that comprehensibility and communication of process's agents around process models is at least as important as their degree of formality (Fuggetta 2000). The use of low-level formalisms by some process description languages, the lack of flexibility and the impossibility for non-programmers to use them, were among the main causes of their limited adoption.

Another fact that became manifest to the software process modeling community was the critical need of having a standard formalism for representing and exchanging software processes. Instead of reinventing the wheel, many industrial and research teams were attracted by the success of UML (Unified Modeling Language) and explored the possibility of using it as a process modeling language (Chou and Chen

2000) (Di Nitto et al. 2002) (OMG SPEM1.0 2002) (Franch and Rib 1998). UML is standard, provides a rich set of notations and diagrams, extension mechanisms and whatever its advantages and drawbacks, it is undeniably one of the most adopted modeling languages of this decade. Experiences with UML were not restricted to the software process community but covered other areas such as the business process and the workflow domains (OMG WFMS 2000). However, these experiences faced in their turn a major barrier. Despite the expressiveness of the language, UML models are not executable. Process models were used as contemplative rather than productive assets. An example of such propositions in the industry is the OMG's SPEM standard (Software Process Engineering Metamodel) (OMG SPEM1.0 2002). While execution was out of the scope of the first version of SPEM (i.e. SPEM1.1), it has been established as a mandatory requirement in its second revision (i.e. SPEM2.0). Unfortunately, the recently adopted standard still fails in ensuring this requirement.

In this paper we propose to deal with the executability issue in the context of UML-based process modeling languages. At this aim, we propose a framework and an approach for modeling and executing software processes. This framework is based on our dedicated language for software process modeling called UML4SPM (UML-based Language for Software Process Modeling) (Bendraou et al. 2005) and an execution support. UML4SPM comes in form of a MOF (Meta Object Facility)-compliant metamodel (OMG MOF 2006), a notation and semantics that extend the UML2.0 standard. For the execution support of UML4SPM, the semantics of the metamodel is defined in terms of operations and instructions in order to form what we call the *Execution Model*. The *Execution Model* is then used as a basis for the realisation of the execution support. In this paper we have experienced two approaches for implementing the *Execution Model*. The first one consists of a Java implementation using the Visitor design pattern. The second one is based on a metaprogramming language called Kermeta (Muller et al. 2005) and the use of aspect oriented modeling techniques. We will discuss both process execution approaches, give advantages and inconvenient of each one and finally discuss the suitability of UML in general for the definition of executable process models. In a previous work (Bendraou et al. 2007), not presented here, we also explored the possibility of transforming UML4SPM process models into BPEL

(Business Process Execution Language) (OASIS BPEL 2007) in order to execute them and we highlighted the limitations of such approach.

It is worth noting that the approach described in this paper for building an executable environment for UML4SPM models (i.e. the *Execution Model*) can be generalised to any other MOF-instance language and is not restricted to UML-based languages.

The paper is organized as follows. Section 2 starts by motivating the use of UML 2.0 as a process modeling language. It highlights its strengths in terms of expressiveness but also its weaknesses such as the lack of an execution support and the inability of the standard to express some primary elements proper to process modeling. To overcome these limitations, an extension to UML is proposed through UML4SPM, our process modeling language. The executability issue is addressed by introducing the notion of the *Execution Model* in Section 3. Two realizations of the *Execution Model* in the context of UML4SPM are presented in section 4. Section 5 discusses the results of these realisations, gives the advantages and limitations of each one and synthesises the outcome of our experimentations. Related work is addressed in Section 6. Finally, section 7 sketches some perspectives and concludes this work.

2 UML as a Basis for Software Process Modeling

In UML2.0, Activities have changed radically from UML1.x. In its version 2.0, UML goes beyond graphical representations by offering a high potential for expressing a large variety of processes (Bendraou et al. 2005). Thanks to *Activity* and *Action* packages, it provides concepts for expressing proactive and reactive controls, conditional branches, loops, exception handling as well as a numerous actions with computational semantics. It also supports a large number of Workflow patterns, a taxonomy of generic, recurring constructs originally devised to evaluate workflow systems, and more recently used to successfully evaluate process modeling and execution languages in general (see (Dumas et al. 2001), (Van der Aalst et al. 2003) and (White 2004)). In accordance with Jablonski and Bussler's original classification (Jablonski and Bussler 1996), these patterns span the control-flow, data and resource perspectives, the two later perspectives being more specific to business processes rather than to software processes. In

(Wohed et al. 2005), authors evaluated the capacity of UML2.0 in modeling twenty control-flow patterns that commonly recur in process models. Examples of such patterns are parallel split, multiple merge, deferred choice, etc. UML2.0 succeeded in representing all of them except for four patterns, which makes it more expressive than some business process formalisms (e.g. BPEL: Business Process Execution Language) (Wohed 2005). Data patterns mainly deal with data visibility, data interaction and data transfer and routing. Examples of such patterns are the multiple instances data pattern, the database task trigger patterns and so on. In (Russel 2005a), it has been demonstrated that eighteen of the forty data patterns were supported by UML2.0, which remains quite satisfactory. As for resource patterns, they address all the issues about work allocation to process's resources, the ability for resources to see the work status, resources allocation conflicts, work distribution and so on. According to (Russel 2005b) however, UML2.0 only satisfies six of the forty-three resource patterns, which reduces its suitability for modeling the resource perspective. Still, many of these perspectives can be addressed at a lower level by the execution support.

All these points, added to the fact that UML is a widely used standard and provides a rich set of notations, make UML a good candidate as process modeling language. However, apart the notion of Activity, it has been demonstrated that UML lacks of some primary process elements, which constitute the vocabulary necessary for modeling software processes (OMG SPEM1.0 2002) (Bendraou et al. 2005). This set of concepts was identified by many initiatives in the literature and regroups elements such as Role, WorkProduct, Agent, Tool, Guidance and Team (Lonchamp 93). In the next section we propose an extension to UML2.0 in order to provide the standard with such concepts. This is done in the context of our language, UML4SPM.

UML4SPM

Our extension, namely UML4SPM, aims in first place at introducing primary process elements to the UML2.0 standard. This is obtained by extending the UML2.0 metamodel and more precisely, the Activity and Artifact metaclasses. This extension comes in form of a MOF-compliant metamodel (OMG MOF 2006) and is presented in fig. 1. White boxes represent the UML metaclasses we extended.

The UML4SPM metamodel aims at defining the minimal subset of concepts for software process modeling while relying on the advanced constructs and activity coordination mechanisms offered by UML2.0.

By making UML4SPM Software Activity extending the UML2.0 Activity metaclass, we take advantage of all its properties and associations. Thus, a Software Activity can be composed of other Software Activities and may contain Actions. An UML2.0 Activity being indirectly a Classifier, the ability to specify new properties and new operations, as well as pre and post conditions on the execution of a Software Activity is also made possible. The UML4SPM WorkProduct element extends UML2.0 Artifact. It represents any physical piece of information consumed, produced or modified during the software development process. An Artifact being a Classifier, WorkProducts can be defined as InputPins and OutputPins of Software Activities and Actions. It is also possible to specify composite WorkProducts thanks to the reflexive "nested artifact" association (not shown in the figure).

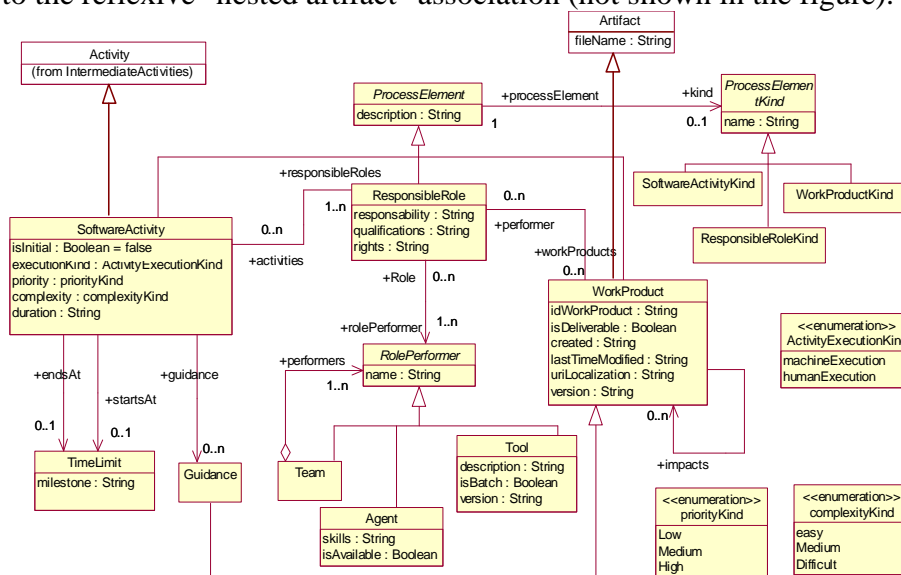


Fig. 1. UML4SPM Metamodel

Since the aim of this paper is to present the executability aspect of UML4SPM and not the language itself, the interested reader can refer to (Bendraou et al. 2005) for more details on the metamodel.

We also enriched the UML2.0 activity diagram notations in order to take into account some new properties and aspects specific to software process modeling that we introduced by our extension. It is important

to note that this extension do not affect neither the comprehensibility of people already familiar with the UML2.0 Activity constructs nor their semantics. One that makes use of Activity diagrams can easily use the UML4SPM notation. This notation is given in fig. 2. Looking to the figure, one can identify the activity's name, its input and output parameters (and possibly their current state), its priority in the process, its duration, the assigned roles, the tools used for performing the activity, accepted and triggered events, if it's machine or human-oriented, etc. Post and pre conditions can be expressed using OCL2.0 constraints (Object Constraint Language). These constraints have to be expressed upon process's constituents (i.e., properties and states of WorkProducts, activities, roles, etc.). Of course, it is not mandatory that all these features appear on the activity representation.

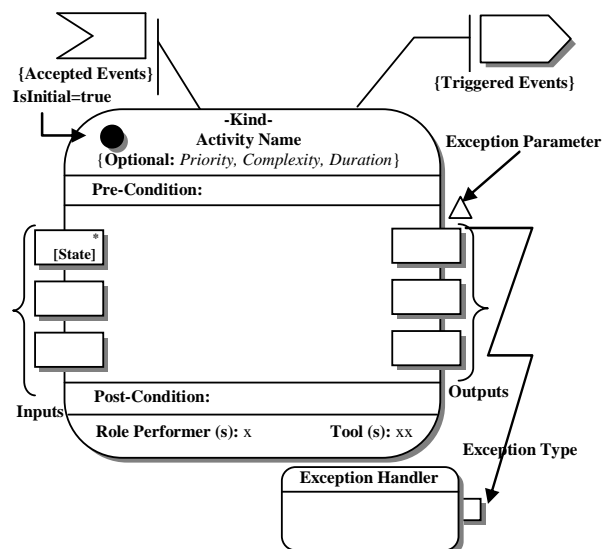


Fig. 2. The UML4SPM Software Activity Notation

Process Example

Fig. 3, gives a simple yet representative example of a portion of a software process modelled using the UML4SPM notation. This process example was provided by our industrial partners within the IST

European Project MODELPLEX². We will use it throughout the paper to demonstrate our approach.

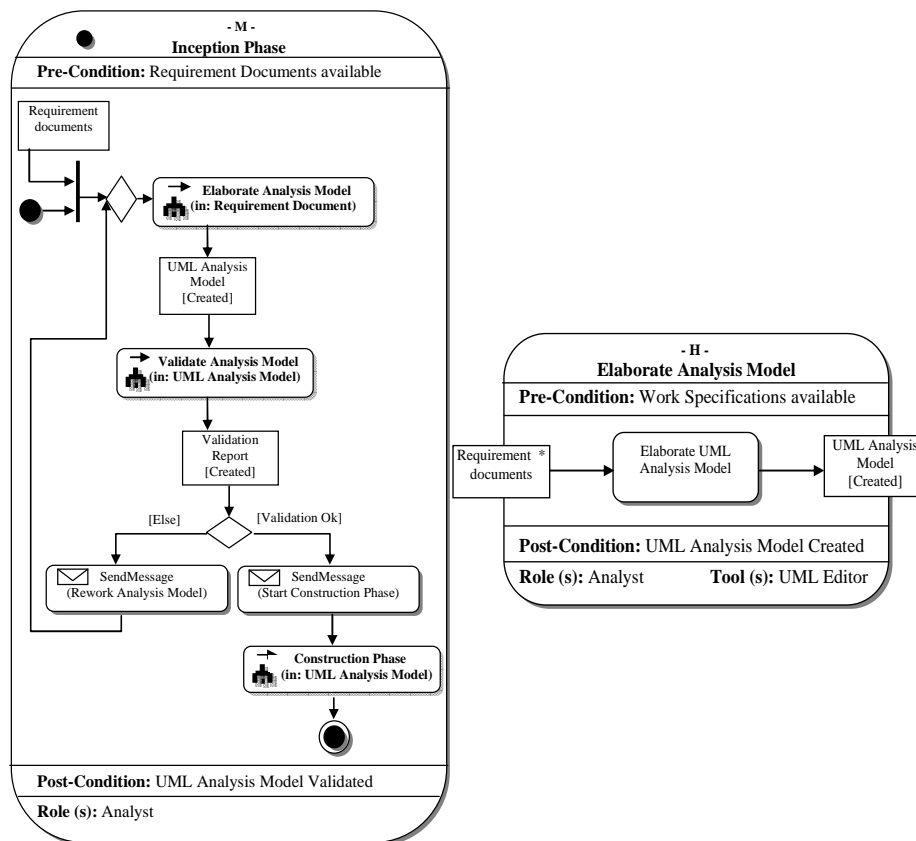


Fig. 3. Software Process Example

The "Inception Phase" activity represents the context of this process (i.e., container for all process's activities). This is indicated by the start-blob in the top-left corner. It is used to coordinate between different process's activities and WorkProducts. The "M" letter is to indicate that the activity is machine-executable (H for Human execution). 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, one has to precise 1) whether the call is synchronous (use of a complete arrow in the top-left corner) or asynchronous (half arrow, e.g., "Construction Phase" call); 2) the parameters of the call, which represent WorkProducts inputs/outputs of the activity. Another aspect

² Modelplex, IST European Project contract IST-3408, at <http://www.modelplex-ist.org/>

is the use of Decision and Merge nodes. The Decision Node allows for the expression of a choice of actions to perform depending on a condition (in this case, if the analysis model is valid or not). Conditions have to be expressed on activity edges (i.e., object flows) and will be evaluated at runtime. The merge node here is used to express that the "Elaborate Analysis model" activity may be triggered by one of the two possibilities. The first one is when the "Inception Phase" activity is launched. The second one is when the analysis model validation fails.

At this level, UML4SPM is used only for modeling purposes. Since it is UML-based, there is no direct support for executing UML models. Even if UML2.0 provides execution semantics for each activity's constructs and actions, no implementation or virtual machine is provided. In the next section, we will see how to deal with this issue by introducing what we call the *Execution Model*. That latter specifies the operational semantics of each element of the UML4SPM metamodel and particularly of the UML2.0 Activity and Action elements. We will then present two realizations of the *Execution Model* as the basis of the UML4SPM's execution support (cf. section 4). The running example described above will be used to explain the approach.

3 The Execution Model

The *Execution Model* tends to bring life to elements of the UML4SPM metamodel. By life, we mean a precise specification of the runtime behaviour of each element of the metamodel. Therefore, a UML4SPM process model once edited can be straightforwardly executed without any additions or intermediate steps. The only condition is that the process model is 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 approach we propose for defining executable models requires two main steps. The first one consists in defining the *Execution Model*,

which aims at specifying how each element of the metamodel should react at runtime and the set of operations it has to perform. In the context of UML4SPM for instance, this means to specify how the activity starts its execution, how roles are assigned to activities, how WorkProducts are automatically routed between activity's actions, how activities react to events, and so on.

The second step is to formalise this execution semantics at the metamodel level. In UML4SPM, the operational semantics was implemented using two different approaches. The first one consisted in implementing the *Execution Model* using Java and the Visitor pattern, the second one by combining a metaprogramming language called Kermeta and aspect modeling techniques.

Execution Model: Rational

The idea of the *Execution Model* is inspired from the RFP (Request For Proposal) issued by the OMG called: Executable UML Foundation (OMG fUML 2009). The objective of this initiative is the definition of a compact subset of UML 2.0 to be known as “Executable UML Foundation”, along with a full definition of its execution semantics. Since that the building blocks of UML4SPM are UML2.0 Activity and Action packages, we found it interesting to take advantage of this specification, while focusing on the UML2.0 elements we reused in our SPML. In UML4SPM, Activity and Action elements are used for sequencing the process's flow of work and data, for expressing actions, events, decisions, concurrency, exceptions, and so on. Thus, the implementation of the execution behavior of these concepts will be used as the core of the UML4SPM engine.

The UML4SPM *Execution Model* comes in form of a class diagram; each class represents the executable semantics of a UML4SPM element. An executable class is a class having a set of operations aiming at describing the execution behavior of the UML4SPM element at runtime. If the element is an UML element reused by UML4SPM, then its semantics is implemented according to the one given in natural language by the UML2.0 standard. The implementation of the UML *Execution Model* was restricted to Activity and Action elements that we reused within UML4SPM, and which respects the UML2.0 semantics (see table 1). Fig. 4. gives an example of the operations and features

required for an Activity Node to execute. In UML, Activity Nodes regroup Actions, Object Nodes (pins), and Control Nodes metaclasses. The execution semantics adopted by UML2.0 activities is quite similar to Petri Nets one and is based on offering and consuming tokens between the different activity's constituents (i.e., Activity Nodes and Activity Edges). This semantics is presented hereunder.

<i>Actions</i>	<i>Activity Elements</i>
- AcceptEventAction	- Activity
- CallBehaviorAction	- Activity Edge (ControlFlow, ObjectFlow)
- CallOperationAction	- Control Nodes (DecisionNode FinalNode, ForkNode, InitialNode, JoinNode, MergeNode)
- RaiseExceptionAction	- ObjectNodes (ActivityParameterNode, Inputpin & Outputpin, DataStoreNode)
- SendSignalAction	- ConditionalNode & LoopNode
- OpaqueAction	- ExceptionHandler

Table 1. UML2.0 activity elements and actions reused in UML4SPM

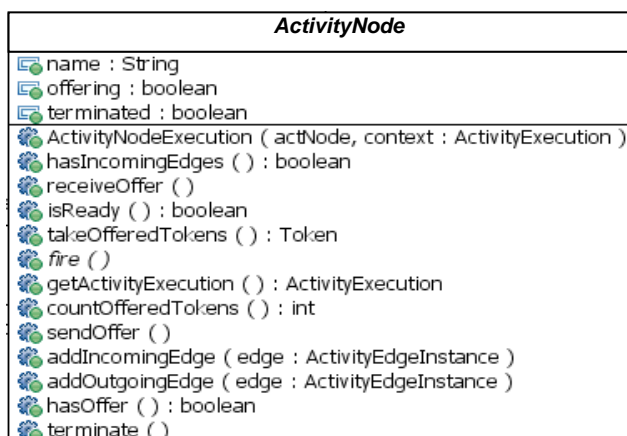


Fig. 4. Specification of the *ActivityNode*'s Behavior

Execution Model: Execution Behavior

In UML2.0, the execution semantics of activities is based on token flows. By flow, we mean that the execution of one activity 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.

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 completes its execution, it creates a control token and offers it to all its outgoing activity edges. 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 completes and if it provides an output, an object token with a reference to the Output Pin type is created. In the context of UML4SPM, an Output Pin can only be typed by WorkProducts or subclasses of the WorkProduct metaclass.

Activity Nodes (i.e. actions, control nodes, etc.) and Activity Edges follow token flow rules as defined by the UML2.0 standard. Activity Nodes control when tokens enter or leave them. Activity Edges have rules about when a token may be taken from the source Activity Node and moved to the target Activity Node. A token traverses an Activity Edge when it satisfies the rules for target and source Activity Nodes, and the Activity Edge, all at once. This means that a source Activity Node can only offer tokens to the outgoing Activity Edges, rather than force them along the Activity Edges, because the tokens may be rejected by the Activity Edges or the target Activity Node on the other side.

Tokens are effectively held by the offering Activity Node until the receiving one is ready to take them. As such mediator, an Activity Edge provides the following functionality: checks whether its source is offering any token, if the guard on the edge is satisfied, send offers of tokens from its source to its target and take the offered tokens from its source to its target Activity Node (see figure 5).

Tokens will be consumed by the executing Activity Node 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 Activity Node (where they will be held up to its consumption), which also sets its offering attribute to true (to indicate that it is now making an offer) and then concurrently calls sendOffer() on all its outgoing edges and, consequently, this will cause each outgoing Activity Edge to call receiveOffer() on its target Activity Node. Figure 5 synthesizes this general execution behavior in form of a UML sequence diagram. It shows all the operations that need to be executed in order to ensure such interactions between any kinds of Activity Nodes.

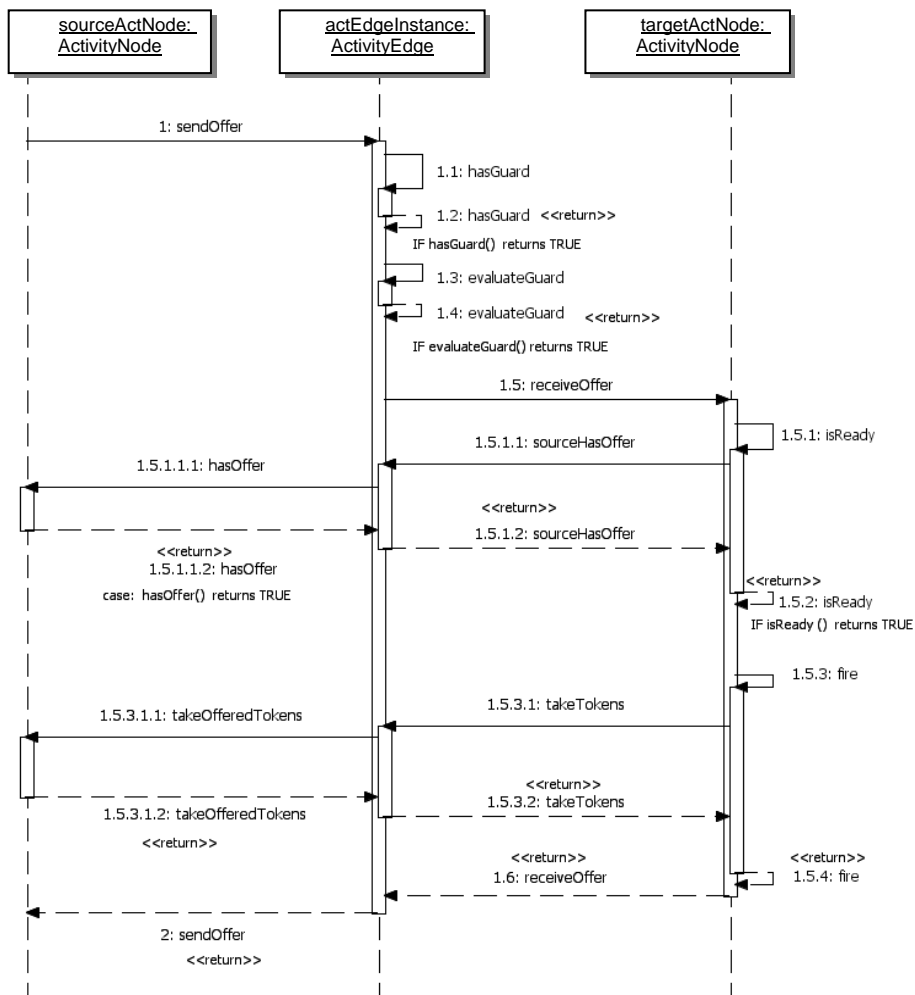


Fig. 5. *ActivityNode* and *ActivityEdge* Interactions

To illustrate this, let's go back to the example we defined in figure 3. When the "Elaborate Analysis Model" action ends, it produces an output, which is the "UML Analysis Model" document. This document is placed in the action's OutputPin. In UML, an OutputPin represents a container that holds action's output values (i.e., Tokens). An action has an OutputPin for each type of output it produces. The same applies for InputPin. This output has then to be consumed by the "Validate Analysis Model" action. Prior to this, the output has to be first put in the action's OutputPin, offered by the OutputPin to all its out coming edges, checked against guards or conditions, if any, which may be specified between the first action's OutputPin and the second action's

InputPin. In the example, we can figure out a guard specifying that the "UML Analysis Model" document's state should be set at "created" when passing from the source action into the target action, otherwise, the target action will not start. If the guard is satisfied and the target action is ready to execute, then the output is transferred from the source action's OutputPin into the target action's InputPin, which would then fire the execution of the action. All these interactions represent an instance of the sequence diagram represented in figure 5.

To refer to the example, it represents the interactions between a source action's OutputPin, the activity edge and a target action's InputPin (see top left side of figure 5). Thus, once all metamodel element's behaviours defined in terms of operations and interactions, which we did in the context of UML4SPM, the next step consist in implementing the *Execution Model*. This is presented in the next section. Of course, these two steps have to be carried only once and are completely transparent to the UML4SPM process modeller, who just instantiates the metamodel (from the graphical editor) and run the process.

4 Realization of the Execution Model

Hereunder we present two realizations of the *Execution Model*.

The Visitor pattern approach with Java

This approach is inspired from the GoF Visitor pattern (Gamma et al. 94). The idea here is to decouple the elements defined in the UML4SPM metamodel from their runtime behavior. Thus, for each element in the UML4SPM metamodel, there is a runtime "Execution" visitor class 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 6 draws the big picture of the *Execution Model* implementation using the Visitor pattern by giving the example of Software Activity, Activity Edge and Activity Node elements and their corresponding executable classes in the *Execution Model*.

In the design of the UML4SPM Executable Model we put as a crucial requirement, to keep 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 for which it defines an 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’s properties 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 still have to be defined. The definition of these conditions is underway and goes beyond the scope of this document.

We provide a Java implementation of this model. This implementation is used as the basis of the UML4SPM process execution Engine. This implementation can be also used as a basis of an activity diagram virtual machine since we implemented the execution behavior of UML2.0 activity and action packages according to the standard.

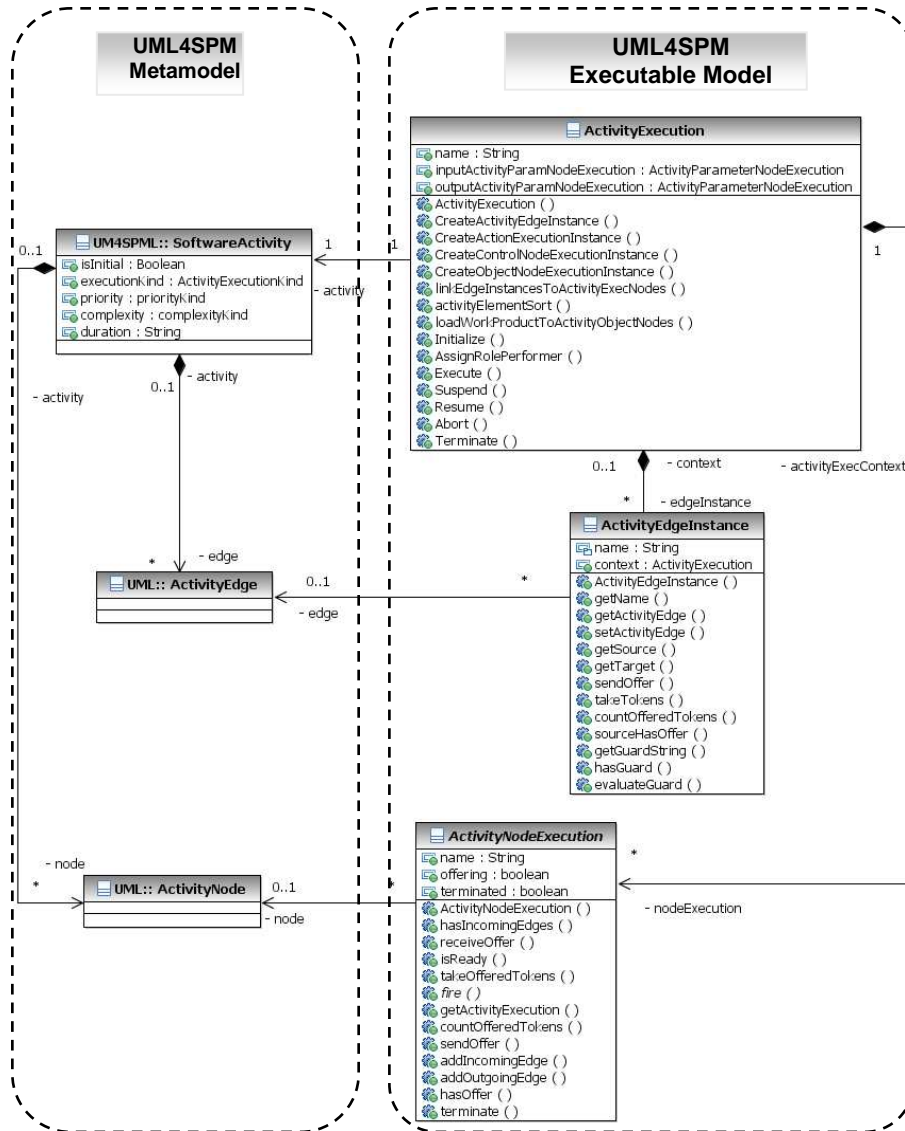


Fig. 6. Execution Model: the visitor approach

The Kermeta approach

Kermeta is an MDE platform designed to specify constraints and operational semantics of metamodels (Muller et al. 2005). The MOF supports the definition of metamodels in terms of packages, classes, properties and operations but it does not include concepts for the

definition of constraints or operational semantics. Kermeta extends MOF with an imperative action language for specifying constraints and operation bodies at the metamodel level.

One of the key features of Kermeta is the static composition operator, which allows extending an existing metamodel with new elements such as properties, operations, constraints or classes. This operator allows defining various aspects in separate units and weaving them automatically into the metamodel. The weaving is done statically and the composed model is typed-checked to ensure the safe integration of all aspects. This mechanism makes it easy to reuse existing metamodels or to split metamodels in reusable pieces. It also provides flexibility. For example, several operational semantics can be defined in separate units for a single metamodel and then alternatively woven depending on a particular need. This is the case for instance in the UML metamodel where several semantics variation points are defined.

The purpose of Kermeta is to remain a core platform for safely integrating all the aspects around a metamodel. For instance, metamodels can be expressed using MOF and constraints using the OCL. Kermeta also allows importing Java classes in order to use services such as file input/output or network communications, which are not available in the Kermeta standard framework. This is very useful for instance to allow interactions between models and existing legacy applications. In the case of UML4SPM, this allows processes to interact with business applications, the enterprise workflow, to call distant web services and so on.

Fig. 7 presents an overview of the architecture of the UML4SPM implementation using Kermeta. The diagram shows the units to be composed in order to build the UML4SPM environment and simulator. Ecore files (UML.ecore and uml4spm.ecore) are metamodels expressed using the Eclipse Modeling Framework (EMF). Because the EMF is compliant with the EMOF standard, these metamodels can be used directly in the implementation. UML.ecore corresponds to the standardized UML 2 metamodel provided by the Eclipse/UML project. The uml4spm.ecore metamodel corresponds to the extension of UML for software process modeling given in Fig. 1.

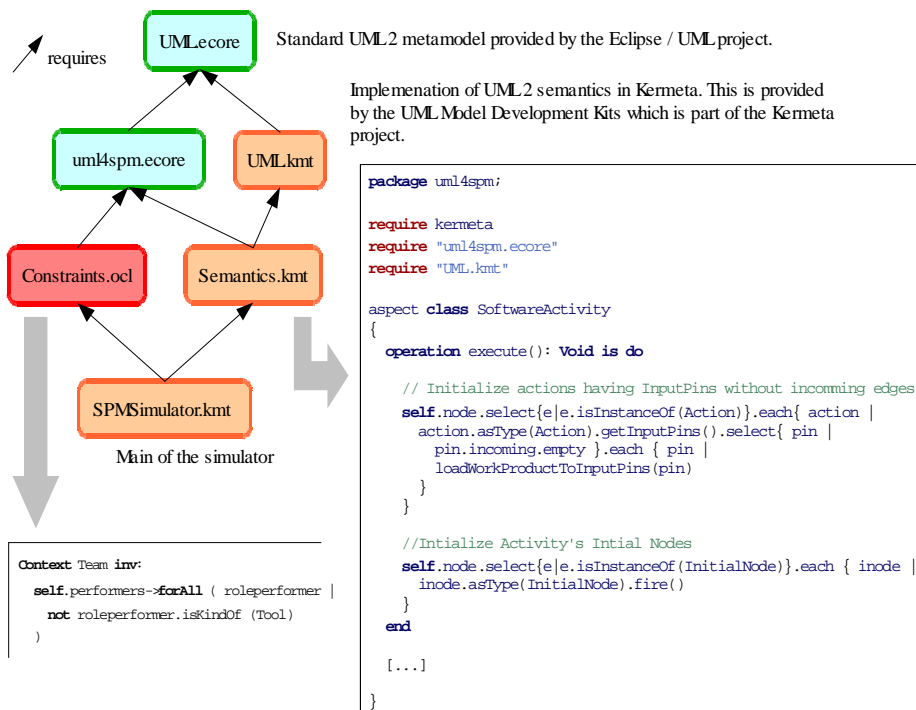


Fig. 7. Weaving Executability to The UML4SPM Metamodel

The *.kmt files on Fig. 7 correspond to Kermeta source files. The UML.kmt is an implementation of the UML semantics in Kermeta. This file especially implements the semantics of UML 2 activity diagrams, which is reused in the context of the UML4SPM extension. The file Semantics.kmt corresponds to the implementation of the UML4SPM *Execution Model*. An excerpt of the source code of this file is shown on the right hand side of Fig. 7. The first line of the listing specifies the containing package for the definition contained in the file. Then the “require” directives are used to declare dependencies with other units. In the example, the uml4spm metamodel defines a metaclass named uml4spm::SoftwareActivity. The piece of code shown on the listing adds an operation named “execute” in this metaclass.

Adding new elements to a metaclass of the metamodel is achieved using the keyword “aspect” before the declaration of the class. The body of the operation “execute” presented in Figure 7 implements how a software activity can be executed. The execution of an activity consists of initializing actions and initial nodes of the activity. In the code, we first search for actions having input pins without incoming

edges in order to initialize them with WorkProducts of the same type and then we look for initial nodes and initialize them by calling the operation “fire”. In order to fully implement the *Execution Model* of the UML4SPM metamodel, all required operations are implemented in the same way as for the “execute” operation detailed on the listing.

The file Constraints.ocl shown in Figure 7 encapsulates constraints on the UML4SPM metamodel. These constraints are written in standard OCL. Figure 7 presents the listing of a simple constraint as an example. In the metamodel given in Figure 1 there is an aggregation called “performers” from the Team metaclass to RolePerformer metaclass. In practice, the performers of a team can be either teams or agents but not tools. The constraint presented is an invariant for the metaclass Team that ensures that no tools can be added as performers.

Finally, the Kermeta source file SPMSimulator.kmt contains the entry point for a simulator, which can load process models (i.e. instances of the uml4spm Ecore metamodel), check the constraints on these models thanks to the OCL constraints and execute these models using operations that were weaved into it.

5 DISCUSSION

Through simulation and execution of software process models, the approach we propose provides project managers with earlier feedbacks on how the process should behave in production stages. It is particularly vital to evaluate process definitions carefully to be sure of their correctness and effectiveness. Important decisions on resource allocation, coordination of agents and procedural issues can then be taken before to put the process on rails. In the following we discuss the outcomes of our work and how they can be used in order to fulfill these expectations.

The Execution Model

Contrarily to traditional process model execution approaches, one key feature of our approach is the ability to execute process models without any transformation or compilation step. Indeed, current propositions require a compilation phase towards some execution

languages, sometimes proprietary, in order to execute them (cf. related work section). This step is most often followed by a manual coding and configuration steps, which is error prone and may induce some traceability issues between process models and their execution. Additionally, these steps have to be performed each time the process definition is modified, which can become a burden for process modelers. Using the *Execution Model* approach, the operational behavior is defined once in the metamodel and can then be instantiated many times. Process modelers do not have to deal with code. It is completely transparent for them. Process models are directly enclosing an execution behavior and can be executed and simulated straightforwardly. Process definitions come in form of UML4SPM models that abstract away all the implementation details and are accessible to a broader community of process users (e.g., engineers, stakeholders, project managers, etc).

A generic approach

In this paper we introduced Executability of models in the context of UML4SPM. However it is worth noting that this approach can be generalized to any MOF-instance language. The same approach can be used for instance to define the execution behaviour of UML state machines in order to make them executable models.

Since the operational semantics we defined respects the one given in the UML2.0 specification, this makes it possible to simulate activity diagrams and to build a UML virtual machine for activity diagrams based on the work presented here. The outcome of the *Execution Model* regarding the UML2.0 elements has been shared with the OMG group working on UML executability in order to provide our feedbacks but also to highlight some new classes, operations or constraints that we defined and which are not addressed by the current OMG's proposition.

Java Vs Kermeta

We provided two realizations of the *Execution Model*: the first one using the Visitor design pattern with Java and the second one using Kermeta and aspect modeling techniques. 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. However two main reasons encouraged us to investigate a more model-driven solution. The first one relates to the fact that the implementation we provide in Java represents one fixed implementation and does not take in charge UML semantic variation points. Indeed, in the UML standard some elements may have different semantics and their implementation is the tool-implementer's responsibility. To give a different Java implementation for each of the semantic variation points and to combine them easily and efficiently would be too complex and error prone. With Kermeta, it is possible to compose (i.e. to weave) different semantics into the metamodel. Process modelers can then choose the appropriate one before starting the process execution. They can also easily extend the behavior of metamodel's elements in order to incorporate new functionalities or simply to take into account some new constraints. It is also possible to define specific kinds of activities that would have in charge the dynamic redefinition of the process model. This would allow the modification of the process at run time in order to take into account for instance new deadline constraints or an unexpected lack of human resources. In Kermeta it is also possible to take into account OCL constraints which are not addressed in Java.

The second reason for choosing Kermeta is because it is more in line with the MDE vision. Indeed, with the Java solution, it is up to the process modeler to code the visitor pattern within the Java classes. This supposes a high knowledge of the UML standard and how metaclasses relate to each others. Using Kermeta, the application of the visitor pattern is completely transparent. The process modeler has only to identify the class for which he/she aims to define a behavior and simply specify it. The Kermeta engine, thanks to aspect techniques, will internally rely metaclasses to each others, will weave their execution behavior and proceed to the execution of their operational semantics.

UML4SPM

In the context of this work, a UML4SPM process model editor and a process engine was provided. The editor is generated automatically

from the UML4SPM metamodel using the EMF Eclipse environment (see figure 8). If the UML4SPM metamodel have to be modified, then the UML4SPM editor have to be 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. Process models are stored using the OMG standard XMI format (OMG XMI 05).

The UML4SPM process execution engine takes as input an UML4SPM process model and executes it according to the execution behavior defined in the UML4SPM executable model. No configuration or intermediate step is required.

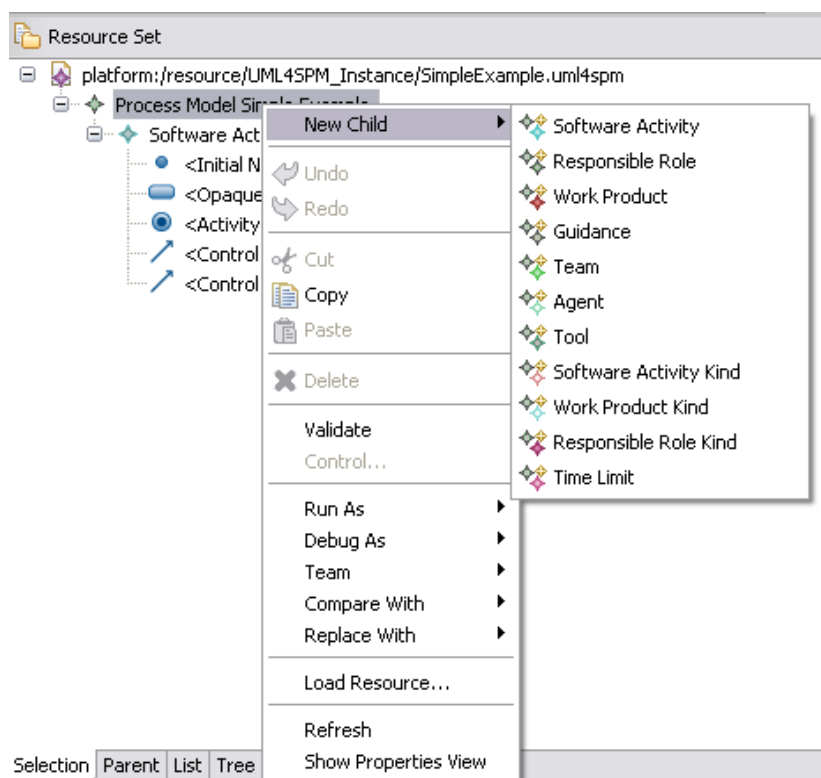


Fig. 8. UML4SPM Process Model Editor

Regarding the expressiveness of UML4SPM, we evaluated it with the well-known ISPW-6 Software Process Example (Kellner et al. 91), a standard benchmark software process problem developed by experts in the field of software process modeling. The description of the

benchmark process by UML4SPM was not just limited to the eight activities of the core problem but it also succeeded to express most optional extensions. Tool invocation actions, communication mechanisms, exception handling, WorkProduct versioning and management features and other constructs offered by UML4SPM were used at this aim. This evaluation is presented in more details in ³.

6 Related Work

In this section we only deal with UML-based process modeling languages, taxonomy of first-generation PMLs can be found in (Zameli and Lee 2001).

In the industrial side, SPEM1.0 was the first standard SPML based on UML (UML1.4) (OMG SPEM1.0 2002). However SPEM1.0 has had a limited success within the industry since SPEM1.0 did not offer any execution support. Process models were only contemplative models. In SPEM2.0, the main advance was the proposition of a clear separation between the content of a method of its possible use within a specific process. SPEM2.0 extends the UML2.0 Infrastructure and does not use any concept from the UML2.0 Superstructure (i.e. Activities, Actions, etc.). Regarding executability, SPEM2.0 does provide neither concepts nor formalisms for executing process models. Instead, the standard proposes to either map process definitions into some project planning tools (e.g. MS. Project) which is not considered as process execution but a process planning activity or to define transformation rules into some business process execution languages (e.g. BPEL). Unfortunately, the standard does not define any of these rules.

In Di Nitto's et al. approach (Di Nitto et al. 2002), authors aim at assessing the possibility of employing a subset of UML1.3 as an executable PML. It comprises two main phases. The first one consists in describing processes using UML diagrams. The second phase consists in translating these UML diagrams into code that can be enacted by the team's events-based workflow engine called OPSS. Process constituents can be defined by simply specializing a set of predefined classes provided by the approach in form of a UML class diagram. The flow of work is given in activity diagrams and the

³ UML4SPM evolution using ISPW6: http://pagesperso-systeme.lip6.fr/Reda.Bendraou/Documents/UML4SPMEvaluation_ISPW6.pdf

lifecycle of each entity is defined by a state machine. However, the activity and class diagrams 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 weak 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 detail how this integration is realized.

Another approach, called Promenade (Franch and Rib 1998), basically follows the same principle as DiNitto's. To model a process, one has 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. 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 description. The approach does not provide any mechanism or way to execute Promenade process models. No tool or prototype was provided.

In (Chou and Chen 2000), Chou proposed a software process modeling language consisting of high-level UML1.4-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. The principal obstacle of this approach is the lack of an automatic generation of process programs from UML diagrams, which imposes the rewriting of the process by developers mastering the proprietary OO language provided by the author.

7 Conclusion

In this paper, we proposed an approach for building executable software process models. Additionally to coding team's best practices,

process models can now be used for simulation and execution purposes. This would help not only for agent coordination, but also can be used as means to improve and to validate process definitions. Executability of models was addressed in the context of a software process modeling language (i.e., UML4SPM) thanks to the *Execution Model* approach. However, it can be generalized to any MOF-instance language. We provided two realizations of the *Execution Model*. The first one is Java using the visitor pattern. The second one, a more model-driven solution, is based on Kermeta and aspect modeling techniques. The outcome of this work is largely used by our industrial partners within the Modelplex projet. A larger evaluation of the use of UML4SPM in production stages is underway. In the context of process modeling, an important perspective of this work is the definition of the set of activities and constraints that would allow a process definition to be modified at runtime and without restarting the process execution i.e. preserving the process state.

References

- 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), Montego Bay, Jamaica, Oct. 2005, LNCS, Vol. 3713, PP 17-38.
- Bendraou, R., Sadovykh, A., Gervais, M.P. and Blanc, X. "Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach". In Proceedings of the 33rd EUROMICRO Conference of Software Engineering Advanced Application (SEAA), pp. 314-321, Lübeck, Germany, IEEE Computer Society Press.
- 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.
- Di Nitto, E. et al. "Deriving executable process descriptions from UML", in Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, FL 2002, ACM Press.
- Dumas, M. and ter Hofstede, A., "UML Activity Diagrams as a Workflow Specification Language," in Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 2001.
- Franch, X. and Rib, J. "A Structured Approach to Software Process Modelling," in Proceedings of the 24th Conference on EUROMICRO - Volume 2, 1998.
- Fuggetta, A. "Software Process: A Roadmap". 22nd International Conference on Software Engineering (ICSE), June 4-11, Limerick (Ireland), ACM, 2000.
- Gamma E., Helm R., Johnson R., and Vlissides J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994.
- Jablonski, S. and Bussler, C. "Workflow Management: Modeling Concepts, Architecture and Implementation", London, UK.: Thomson Computer Press, 1996.

- 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 Proc. of the first Intern. Conf. on the Software Process. IEEE Computer Society, Washington, DC, 1991, pp. 176-186.
- Lonchamp, L. "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, CA., USA, 1993.
- Muller, P.A. Fleurey, F. and Jézéquel, J.M. "Weaving executability into object-oriented meta-languages" In Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pp 264-278, Montego Bay, Jamaica, October 2005. Springer-Verlag.
- OASIS, Web Services Business Process Execution Language Version 2.0. Working Draft. WS-BPEL TC OASIS, January 2007.
- OMG MOF, "Meta Object Facility version 2.0", adopted specification, OMG document formal/06-01-01, January 2006, at <http://www.omg.org>.
- OMG, "Semantics of a Foundational Subset for Executable UML Models RFP", OMG document ad/05-04-02, May 2008, at: <http://www.omg.org/docs/ad/05-04-02.pdf>
- OMG SPEM1.0, "Software Process Engineering Metamodel", OMG document formal/02-11/14, November 2002, at <http://www.omg.org>.
- OMG, "Workflow Management Facility Specification v1.2", OMG document formal/00-05-02, April 2000, at <http://www.omg.org>.
- OMG XMI, "XML Metadata Interchange", version 2.1., OMG document formal/05-09-01 , September 2005 at <http://www.omg.org>
- Osterweil, L. "Software Processes Are Software Too" in Proceedings of the 9th International Conference on Software Engineering (ICSE'9), New York, 1987, ACM Press.
- Russell, N. Ter Hofstede, A., Edmond D. et al., "Workflow data patterns: Identification, representation and tool support," in Proceedings of the 25th International Conference on Conceptual Modeling, Klagenfurt, Austria, 2005.
- Russell, N., Van der Aalst, W. et al., "Workflow resource patterns: Identification, representation and tool support," in Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005, pp. 216-232.
- Van der Aalst, W. et al. "Workflow Patterns", in journal of Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.
- White, S. "Process modeling notations and workflow patterns", Workflow Handbook 2004, L. Fischer, ed., pp. 265-294, FL, USA: Future Strategies Inc., Lighthouse Point, 2004.
- Wohed, P. et al. "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 (a).
- Wohed, P. et al., "Pattern-based analysis of UML activity diagrams," in Proceedings of the 25th International Conference on Conceptual Modeling (ER'2005), Klagenfurt, Austria, 2005 (b).
- Zameli, K. Z., Lee, P.A. "Taxonomy of Process Modelling Languages", in Proc. of the ACS/IEEE Inter. Conf. on Computer Systems and Applications (AICCSA'01) Beirut, Lebanon, June 2001.