Combining Aspect and Model-Driven Engineering Approaches for Software Process Modeling and Execution[*]

Reda Bendraou[1], Jean-Marc Jezéquél[2,3], and Franck Fleurey[4]

[1] University Pierre & Marie Curie
4, Place Jussieu, Paris F-75005, France
{firstname.lastname@lip6.fr}
[2] INRIA-Rennes Bretagne Atlantique, Campus de Beaulieu
F-35042 Rennes Cedex, France
{firstname.lastname@inria.fr}
[3] IRISA, Université Rennes 1
Campus de Beaulieu
F-35042 Rennes Cedex, France
[4] 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 Kermeta, an executable metaprogramming language.

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

1 Introduction

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 artifacts 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.

[*] This work is supported in part by the IST European project "MODELPLEX" (contract no IST-3408).

During the last two decades, the need for executable Software Process Modeling Languages (SPML) has been widely recognized. Osterweil opened the way with its seminal work *"Software Processes are Software Too"* [12]. 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 [4]. 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 [2] [3] [10] [15]. 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 [9]. 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) [10]. 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 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. The proposed framework is based on our dedicated language for software process modeling called UML4SPM (UML-based Language for Software Process Modeling) [1] and a metaprogramming language called Kermeta [7]. UML4SPM comes in form of a MOF (Meta Object Facility)-compliant metamodel [11], a notation and semantics that extend the UML2.0 standard. To make UML4SPM process models executable, the semantics of the metamodel is implemented in terms of operations and instructions using Kermeta. This implementation is then woven into the UML4SPM metamodel using aspect techniques. It is worth noting that the approach described in this paper for building an executable environment for UML4SPM models 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 discusses how UML 2.0 *Activities* can be extended to build a software process modeling language and details the

UML4SPM language. Section 3 presents the executable semantics of UML4SPM and shows how it is implemented using Kermeta. An example is used to illustrate our approach. Related work is addressed in Section 4. Finally, in section 5 we discuss this work and we conclude the paper.

2 UML as a Basis for Software Process Modeling

In UML2.0, *Activities* have changed radically from UML1.x. Indeed, in the last version of the standard, *Activities* are not only suitable for modeling processes; they also have some features to support their automation. This is made possible thanks to *Action* packages, which now allow expressing the semantics of most executable instructions that one can find in common programming languages.

UML2.0 *Activities* also provide coordination mechanisms in order to ensure *proactive control*[1] and *reactive control*[2]. The first kind of coordination mechanism is ensured using concepts such as *Control Flow*, *Object Flow* and *Invocation Actions* (e.g., *CallBehaviorAction*). Reactive control is ensured thanks to the use of UML2.0 *Events*, *AcceptEventAction* and *SendSignalAction* constructs. For more sophisticated coordination mechanisms like concurrency, synchronization, merge, etc., *Control Nodes* can be employed. For instance, a *Fork Node* combined with a *CallBehaviorAction* can be used for modeling multiple and parallel activity calls. Furthermore, some experiences have been realised in order to evaluate the ability of UML2.0 *Activities* to support some well-known and complex Workflow patterns [13]. These experiences revealed that UML2.0 supports more than thirty control flow patterns of forty-three, which makes it more expressive than most business process formalisms such as BPEL (Business Process Execution Language) [14]. UML2.0 also offers some advanced constructs such as *Loop*, *Conditional Nodes*, and concepts to deal with exception handling, which is lacking in most current SPML propositions. All these facilities offered by UML2.0 added to the fact that it is a standard, that many people are already familiar with its notation and diagrams, and that a wide bunch of tooling support is provided, make UML a good candidate as a software process modeling language [1]. However, apart from the notion of *Activity*, UML lacks of some primary process elements, which constitute the vocabulary necessary for modeling software processes. This set of concepts was identified by many initiatives in the literature and regroups elements such as *Role*, *WorkProduct*, *Agent*, *Tool*, *Guidance*, *Team*, etc. [6].

In our proposition UML4SPM, we propose to deal with this issue by introducing these primary process elements into UML2.0. This is obtained by extending the UML metamodel and more precisely, the *Activity* and *Artifact* metaclasses. This extension comes in form of a MOF-compliant metamodel and is presented in fig. 1. White boxes represent the UML metaclasses we extended, i.e. UML2.0 *Activity* and *Artifact* metaclasses.

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. Since the aim of this paper is to

[1] An imperative specification of the order in which activities (actions) are to be executed - direct invocation.

[2] A reactive specification of the conditions or events in response to which activities (actions) are to be executed - indirect invocation.

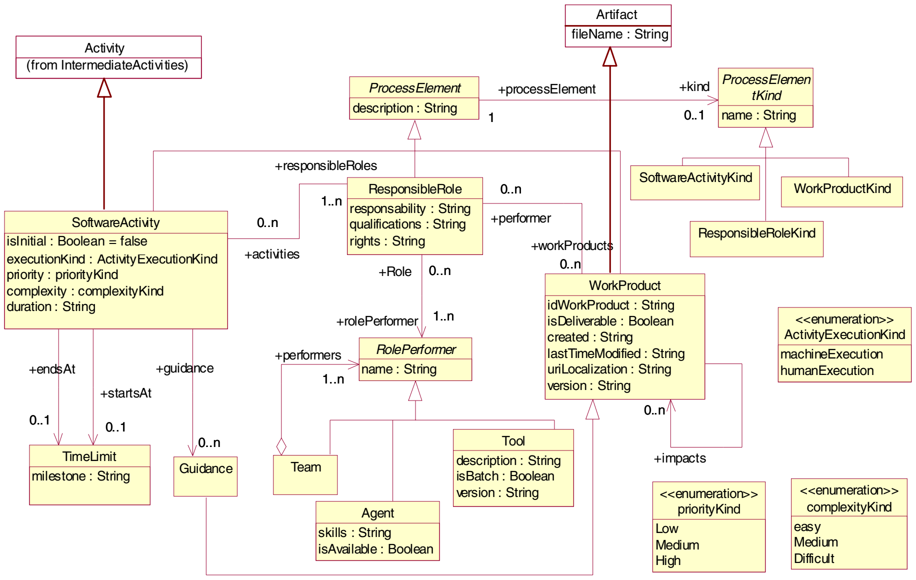


Fig. 1. UML4SPM Metamodel

present the executability aspect of UML4SPM and not the language itself, the interested reader can refer to [1] for more details on the metamodel.

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 presented in the figure).

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 notations. 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

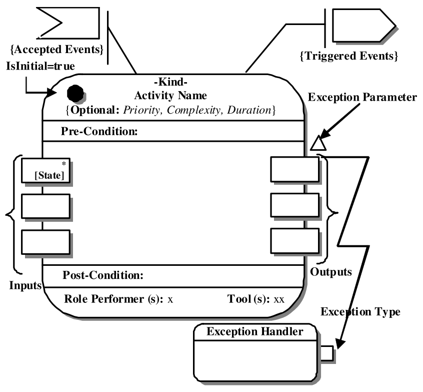


Fig. 2. The UML4SPM Software Activity Notation

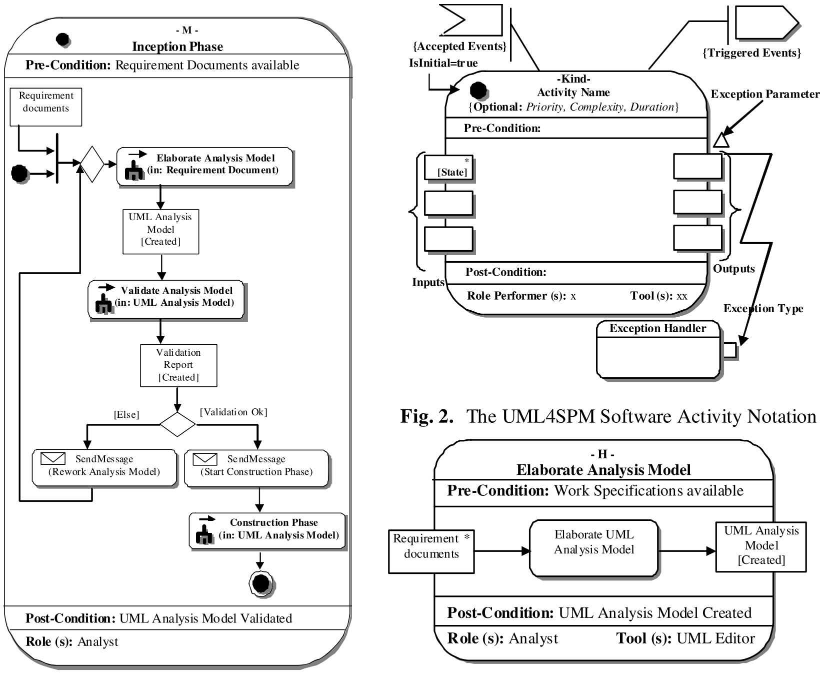

- M -
Inception Phase
Pre-Condition: Requirement Documents available
Requirement documents
Elaborate Analysis Model (in: Requirement Document)
UML Analysis Model [Created]
Validate Analysis Model (in: UML Analysis Model)
Validation Report [Created]
[Else]
[Validation Ok]
SendMessage (Rework Analysis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

Fig. 3. Software Process Example

course, it is not mandatory that all these features appear on the activity representation. 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 MODELPLEX[3]. We will use it throughout the paper to demonstrate our approach.

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 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

[3] Modelplex, IST European Project contract IST-3408, at http://www.modelplex-ist.org/

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 *Execution Model*. That latter specifies the operational semantics of each element of the UML4SPM metamodel and particularly of UML2.0 *Activity* and *Action* elements. The *Execution Model* is then implemented using Kermeta, our metaprogramming language. The running example described above will be used to explain the approach.

3 Weaving Executability into Metamodels

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 the operational semantics of the metamodel. It defines 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, etc.

The second step is to formalise this semantics at the metamodel level. In UML4SPM, the operational semantics was implemented using Kermeta and integrated to the metamodel. The following sub-sections present the UML4SPM *Execution Model* and its implementation using Kermeta.

3.1 Definition of the Execution Model

The idea of the *Execution Model* is inspired from the RFP (Request For Proposal) issued by the OMG called: *Executable UML Foundation* [8]. 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 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 engine of UML4SPM.

The UML4SPM *Execution Model* introduces the execution model in form of class diagrams; each class represents the executable class 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 by the UML2.0 standard in natural language. 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.

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 (pin*s), 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*).

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.

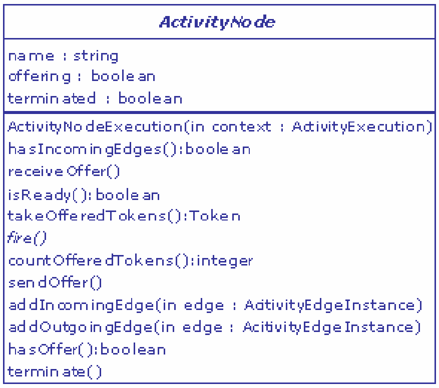

ActivityNode
name : string
offering : boolean
terminated : boolean
ActivityNodeExecution(in context : ActivityExecution)
hasIncomingEdges():boolean
receiveOffer()
isReady():boolean
takeOfferedTokens():Token
fire()
countOfferedTokens():integer
sendOffer()
addIncomingEdge(in edge : ActivityEdgeInstance)
addOutgoingEdge(in edge : AcitivityEdgeInstance)
hasOffer():boolean
terminate()

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

Although the example looks very simple in the figure, in order to execute, many actions have to be carried out. Each concept has a precise behaviour to perform. Fig. 5. shows a sequence diagram that generalizes all the operations that need to be executed in order to ensure such interactions between any kind of *Activity Nodes*. To refer to the example, it represents the interactions between the source action's outputpin, the activity edge and the target action's inputpin.

Thus, once all metamodel element's behaviours defined in terms of operations, the next step consist in implementing them using Kermeta and to weave them as *aspects* into the UML4SPM metamodel. 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 a graphical editor for instance).

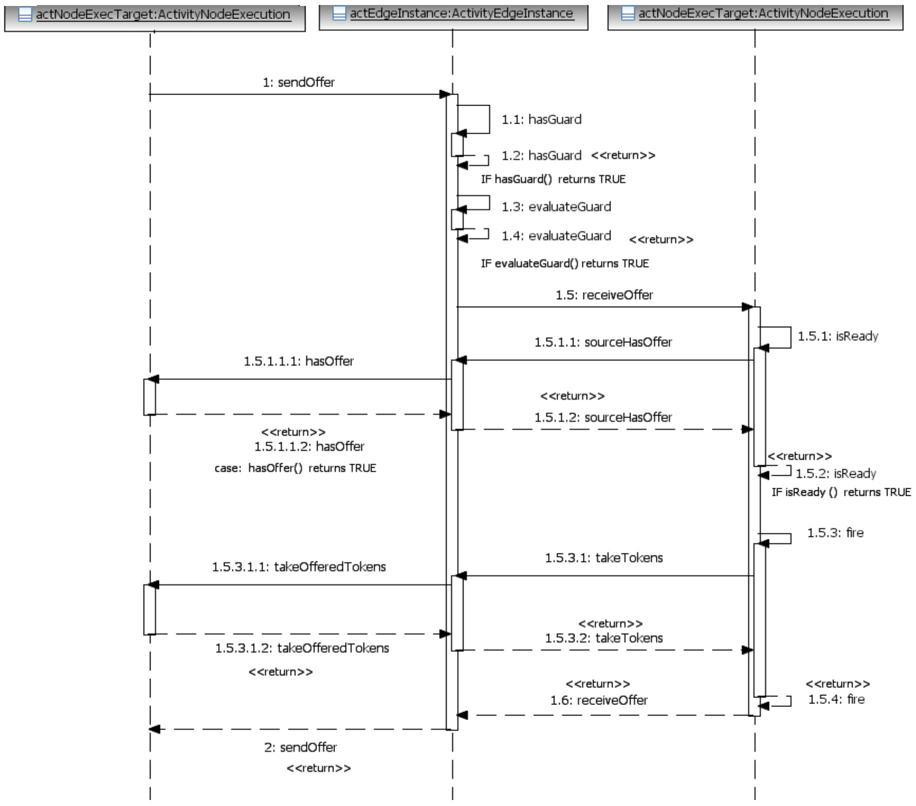


Fig. 5. *ActivityNode* and *ActivityEdge* Interactions

3.2 Implementation of the Execution Model Using Kermeta

Kermeta is an MDE platform designed to specify constraints and operational semantics of metamodels [7]. The MOF [11] 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 Java 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.

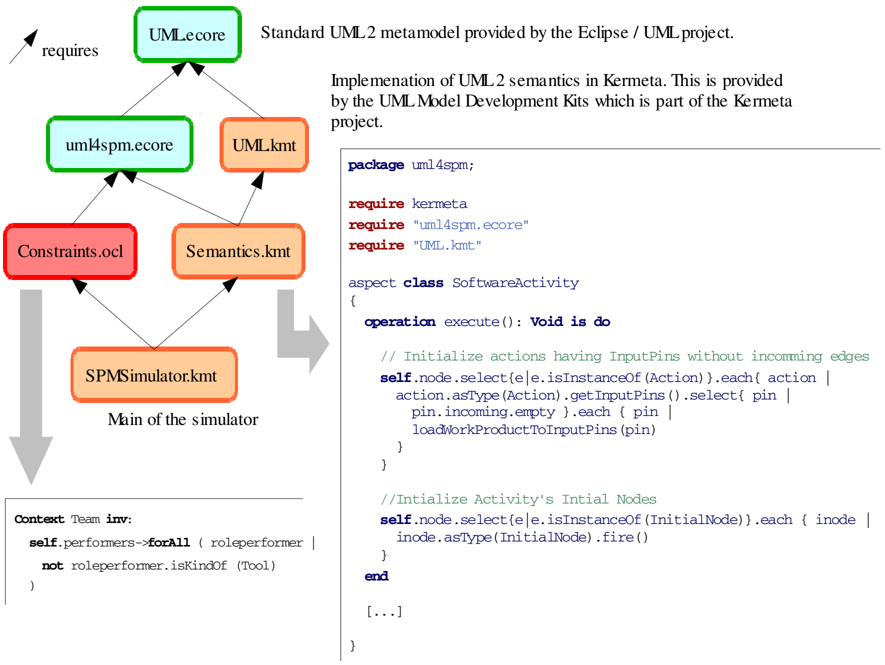
Fig. 6 presents an overview of the architecture of the UML4SPM implementation using Kermeta. The diagram shows the units to be composed in order to build the UML4SPM environment and simulator. Ecore files (UML.ecore and uml4spm.ecore) are metamodels expressed using the Eclipse Modeling Framework (EMF). 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.

The *.kmt files on Fig. 6 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. 6. 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 6 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 6 encapsulates constraints on the UML4SPM metamodel. These constrains are written in standard OCL. Figure 6 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.


requires
UMLecore
Standard UML2 metamodel provided by the Eclipse / UML project.
Implemenation of UML2 semantics in Kermeta. This is provided by the UML Model Development Kits which is part of the Kermeta project.
uml4spm.ecore
UML.kmt
Constraints.ocl
Semantics.kmt
SPMSimulator.kmt
Main of the simulator

```
package uml4spm;

require kermeta
require "uml4spm.ecore"
require "UML.kmt"

aspect class SoftwareActivity
{
  operation execute(): Void is do

    // Initialize actions having InputPins without incomming edges
    self.node.select{e|e.isInstanceOf(Action)}.each{ action |
      action.asType(Action).getInputPins().select{ pin |
        pin.incoming.empty }.each { pin |
        loadWorkProductToInputPins(pin)
      }
    }

    //Intialize Activity's Intial Nodes
    self.node.select{e|e.isInstanceOf(InitialNode)}.each { inode |
      inode.asType(InitialNode).fire()
    }
  end

  [...]

}
```

```
Context Team inv:
  self.performers->forAll ( roleperformer |
    not roleperformer.isKindOf (Tool)
  )
```

Fig. 6. Weaving Executability to The UML4SPM Metamodel

4 Related Work

In this section we only deal with UML-based process modeling languages, taxonomy of first-generation PMLs can be found in [16].

In the industrial side, SPEM1.0 was the first standard SPML based on UML (UML1.4) [10]. 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 [3], 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 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 [15], 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 [2], 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.

5 Discussion and Conclusion

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. section 4). This step is most often followed by a manual coding step for configuring some aspects of the process execution, which is error prone and may induce some traceability issues between process models and their execution. Using Kermeta, the execution 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 without any compilation or transformation phase.

It is also worth noting that the operational semantics we defined respects the one given by the UML2.0 specification. The fact that that latter is weaved into the UML2.0 metamodel makes it possible to simulate UML2.0 activity diagrams. Since UML4SPM extends UML2.0, this semantics is used as the building block of the

UML4SPM simulator. Kermeta also offers features that allow triggering actions outside the Kermeta virtual machine. This would allow the process execution to interoperate with enterprise's applications or external services.

Regarding the expressiveness of UML4SPM, we evaluated it with the well-known ISPW-6 Software Process Example [5], 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 [4] .

Finally, in this paper we introduced *Executability* of models in the context of UML4SPM, however, it can be generalized to any MOF-instance language. 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. This work is ongoing using Kermeta and aspect oriented modeling techniques.

References

1. Bendraou, R., Gervais, M.-P., Blanc, X.: UML4SPM: A UML2.0-based metamodel for software process modelling. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 17–38. Springer, Heidelberg (2005)
2. Chou, S.C., Chen, J.Y.J.: Process Program Development Based on UML and Action Cases, Part 1: the Model. Journal of Object-Oriented Programming 13(2), 21–27 (2000)
3. Di Nitto, E., et al.: Deriving executable process descriptions from UML. In: Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Fl. ACM Press, New York (2002)
4. Fuggetta, A.: Software Process: A Roadmap. In: 22nd International Conference on Software Engineering (ICSE), Limerick (Ireland), June 4–11. ACM, New York (2000)
5. 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, pp. 176–186. IEEE Computer Society, Washington (1991)
6. Lonchamp, J.: 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 (1993)
7. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented metalanguages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
8. OMG, Semantics of a Foundational Subset for Executable UML Models RFP, OMG document ad/05-04-02 (April 2005), http://www.omg.org/docs/ad/05-04-02.pdf
9. OMG, Workflow Management Facility Specification v1.2, OMG document formal/00-05-02 (April 2000), http://www.omg.org

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

10. OMG SPEM1.0, Software Process Engineering Metamodel, OMG document formal/02-11/14 (November 2002), `http://www.omg.org`
11. OMG MOF, Meta Object Facility version 2.0, adopted specification, OMG document formal/06-01-01 (January 2006), `http://www.omg.org`
12. Osterweil, L.: Software Processes Are Software Too. In: Proceedings of the 9th International Conference on Software Engineering (ICSE 9). ACM Press, New York (1987)
13. Van der Aalst, W.M.P., et al.: Workflow Patterns. Journal of Distributed and Parallel Databases 14(3), 5–51 (2003)
14. Wohed, P., et al.: Pattern-based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 63–78. Springer, Heidelberg (2005)
15. Franch, X., Rib, J.: A Structured Approach to Software Process Modelling. In: Proceedings of the 24th Conference on EUROMICRO, vol. 2 (1998)
16. 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 2001), Beirut, Lebanon (June 2001)