Thomas Kühne
Bran Selic
Marie-Pierre Gervais
François Terrier (Eds.)

LNCS 6138

Modelling Foundations and Applications

6th European Conference, ECMFA 2010
Paris, France, June 2010
Proceedings

Springer

eSPEM – A SPEM Extension for Enactable Behavior Modeling

Ralf Ellner[1], Samir Al-Hilank[2], Johannes Drexler[2], Martin Jung[2], Detlef Kips[1,2], and Michael Philippsen[1]

[1] University of Erlangen-Nuremberg, Computer Science Department 2, Programming Systems Group, Martensstr. 3, 91058 Erlangen, Germany
{ralf.ellner,philippsen}@cs.fau.de

[2] Develop group Basys GmbH, Am Weichselgarten 4, 91058 Erlangen, Germany
{alhilank,drexler,jung,kips}@develop-group.de

Abstract. OMG's SPEM – by means of its (semi-)formal notation – allows for a detailed description of development processes and methodologies, but can only be used for a rather coarse description of their behavior. Concepts for a more fine-grained behavior model are considered out of scope of the SPEM standard and have to be provided by other standards like BPDM/BPMN or UML. However, a coarse granularity of the behavior model often impedes a computer-aided enactment of a process model. Therefore, in this paper we present eSPEM, an extension of SPEM, that is based on the UML meta-model and focused on fine-grained behavior and life-cycle modeling and thereby supports automated enactment of development processes.

Conventions in This Paper

Names of meta-classes, packages, and properties are printed in *italics*, names of model elements in `monospaces`, and names of a model element's meta-class are **boldface**. Whenever we refer to SPEM or UML without explicitly specifying a version, we mean SPEM Version 2.0 [1] and UML Version 2.2 [2].

The figures in this paper show meta-elements (e.g., meta-classes, packages, and associations) initially defined in SPEM or UML and their instances (model elements) with a white background and thin lines. Meta-elements that are introduced by eSPEM and their instances are shown with a light gray background and thicker lines. Figures showing parts of the eSPEM refer to the merged, flat meta-model and therefore do not show qualified names. For sake of readability, attributes, operations, and constraints of meta-classes are omitted in the figures unless we refer to them.

SPEM clearly distinguishes between processes and methods. This is reflected in the meta-model with different meta-classes for identical or similar concepts (e.g., *TaskDefinition* for a method and *TaskUse* for a process). Whenever we do not specifically name a meta-class (e.g., *Task* instead of either *TaskDefinition* or *TaskUse*), we are referring to both the method and the process meta-classes.

T. Kühne et al. (Eds.): ECMFA 2010, LNCS 6138, pp. 116–131, 2010.

1 Introduction

Developing software systems in international teams spanning several organizations requires well-defined software development processes (SDP) [3,4]. In order to efficiently define and execute SDPs, one needs 1) a process modeling language (PML) that is rich enough for automatic SDP enactment, 2) an easy-to-use process modeling environment (PME) that is flexible enough for different project categories, and 3) a process execution environment (PEX) that can be adapted to be integrated into existing development tool chains. To be useful for practitioners, there are some well-known requirements PMLs must fulfill [5,6,3]. Exploiting our numerous experience from real-life development projects, we enhance these PML requirements gaining the following list of requirements that a well-integrated combination of PML, PME, and PEX has to fulfill:

- Scalability. The PML used to describe an SDP must work for large as well as small processes.
- Decomposability. Subprocesses and their interfaces with compound processes can be defined.
- Adaptability. Tailoring a given process description to the needs of a defined project must be straightforward.
- Testability. Plausibility checks can be performed automatically on a process description to help in designing SDP models, to supervise their enactment, and to ease CMMI and SPICE auditing [7,8].
- Easy-to-digest formalism. An easy-to-digest formalism is needed because in the past complex formalisms have prevented fine-grained development process models from being adopted by practitioners (see for example [3]).
- Executability. The PML can be directly interpreted by a machine or otherwise mapped to another executable language.
- Automatic process enactment. Using the SDP model formulated in a PML as an input, the PEX supports and guides stakeholders in their work according to the process. It triggers certain activities on time, e.g., it invokes the tools the developers need to open artifacts, and controls the delivery of artifacts.
- Integration. Tools that are already used by developers must be integrated into the PEX.
- Electronic process guide. The SDP documentation actively guides developers by providing information that is sensitive to the task context at hand.
- Automatic audit trail. The PEX automatically keeps track of changes to artifacts as well as progress and conformance of a project with respect to the SDP.

Many of the PMLs known from literature can only partially fulfill the requirements listed above. SPEM (Software and Systems Process Engineering Metamodel) defined by the OMG [1] constitutes a promising approach. As SPEM is based on UML Infrastructure and defines a graphical notation, it is easy to pick up by practitioners, and is considered an ideal basis for SDP modeling. SPEM adequately fulfills the mentioned requirements for a PML except one: executability [9].

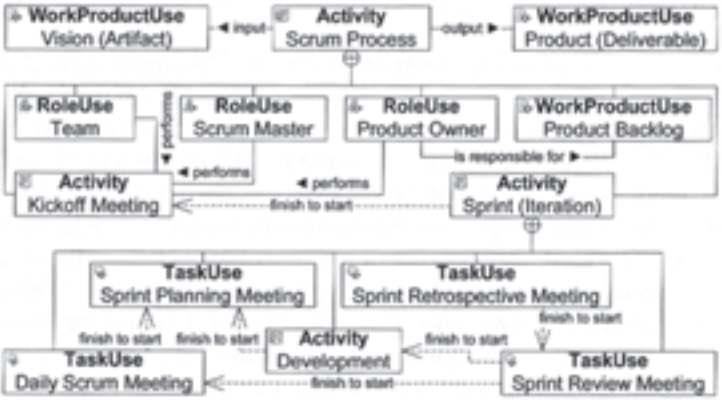

Fig. 1. A SPEM model for the WBS of Scrum

To understand what is required for executability, consider Fig. 1 that demonstrates how SPEM can be used to describe the static structure of an SDP. It shows – with some omissions due to readability – the work breakdown structure (WBS) of the Scrum [10] SDP. However, the problems shown in this example are not specific to Scrum but apply to SDP modeling with SPEM in general.

Many static SDP concepts can be easily expressed in SPEM: *Roles* like `Team` or `Scrum Master` may perform work like the *Activity* `Kickoff Meeting`. *Activities* may have *WorkProducts* as parameters, may be decomposed into sub-activities, and contain further SDP elements. The basic execution order of *Activities* and *TaskUses* is given by precedence edges (e.g., `finish to start`). *Roles* may also be responsible for certain *WorkProducts*. When considering the requirements listed above, there are at least three types of problems:

First, SPEM does not provide its own behavior modeling approach but leaves the integration of behavior modeling languages up to implementers of the specification [1, Sect. 10, p. 69]. However, without fine-grained behavior modeling concepts for proactive and reactive control, no reasonable support can be provided by a PEX that enacts the process. The SPEM standard suggests UML (*Activities* and *StateMachines*) [2] or BPDM/BPMN [11] as candidates. For interfacing behavior modeling languages SPEM provides several generic meta-classes as depicted in Fig. 2.

In our approach we use the behavior modeling concepts from UML. Their notation is common and therefore easy to pick up by practitioners. It includes constructs that are necessary for a precise behavior modeling of SDPs (e.g., *StateMachines*) but are not available in BPDM/BPMN. Although UML does provide the constructs required for a fine-grained modeling of the behavior of SDPs (e.g., asynchronous events and decisions), the SPEM concept for interfacing with behavior modeling languages is cumbersome to use.

Considering Fig. 2, it is obvious that *Roles* may trigger *Transitions* but cannot directly trigger *Events* or execute *Actions*. Consequently, when using this

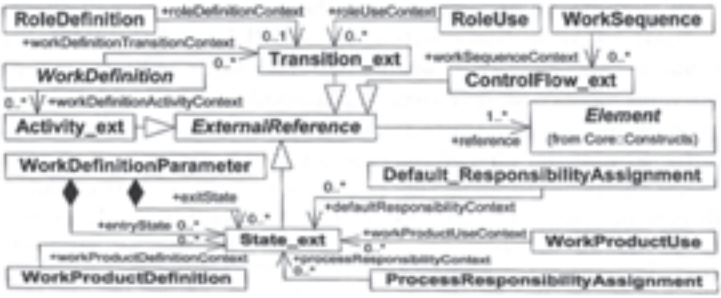

Fig. 2. Behavior model integration in SPEM package *ProcessBehavior* [1, Fig. 10.1]

behavior interfacing mechanism from SPEM, constructs for reactive control (e.g., events and decisions) cannot be sufficiently integrated into the SDP model.

SPEM uses a set of *States* to describe the life-cycle of a *WorkProduct* instead of a *StateMachine*. This contradicts our requirement of an easy-to-digest formalism because it complicates modeling and requires additional consistency rules. In order to avoid these drawbacks of the proposed behavior integration concept from SPEM, we will provide our own solution, combining the advantages of the static process structure modeling features offered by SPEM with the convenient behavior modeling concepts of UML.

Second, experience in industrial development projects shows that in most cases the *Tasks* performed throughout a process are not completely known in advance, but have to be determined and planned when the process is already being executed (e.g., [10]). Another problem exists: Creating, planning, and executing *Tasks* often happens at different points during a process (e.g., in Scrum tasks of a sprint are created and planned in the `Sprint Planning Meeting` and executed during `Development`). In SPEM *Tasks* are always instantiated and executed right away. Thus, the stakeholders cannot be sufficiently supported when planning or working with dynamically created *Tasks*.

Third, modern agile SDPs tend to be used with a wide variety of different development methods without changing the process itself (e.g., the `Sprint Planning Meeting` in Scrum can be used with many different estimation methods). SPEM supports a separation between process and methods using meta-classes that act as proxies for meta-classes representing method elements (e.g., *TaskUse* is a proxy for a *TaskDefinition*). However, a proxy may reference at most one method element. Consequently, changing the target of the proxy also requires changing the process part of the model using SPEM's tailoring concept and raises the modeling effort to adapt a SPEM-based process to a project's needs.

Due to these three issues (behavior modeling support, planning support, and configuration support) many SDPs (e.g., Scrum, Open Unified Process (OpenUP) [12], and V-Modell®XT [13]) cannot be sufficiently modeled with SPEM. Moreover, without such a fine-grained modeling of SDP concepts the stakeholders cannot be sufficiently guided and supported by a PEX that uses

the SDP model as its input. We address these shortcomings in this paper and present eSPEM, a CMOF-based extension of SPEM and the UML Superstructure, that enables the mentioned automatic enactment of SDPs and adequately fulfills all requirements listed above. Additionally, eSPEM also provides solutions for other issues, e.g., instance feature support (*Properties* and *Operations*), enhanced *Kinds*, and detailed *WorkProduct* structure modeling. However, we cannot discuss these solutions in this paper due to space restrictions.

The subsequent paper is organized as follows: Section 2 addresses the three main issues of SPEM and presents our solutions. In Sect. 3, we apply eSPEM to SDPs used in industry, followed by a detailed comparison to other existing approaches in Sect. 4. In Sect. 5, we conclude with a brief outlook at the next steps of our project towards a fully automated enactment of SDP models.

2 Proposed Solution

First, we will show how we substitute behavior interfacing concepts of the original SPEM with our more fine-grained approach. We will then present the additional extensions introduced to enhance SDP enactment support. For complexity reasons, the figures presented in this section only reflect parts of eSPEM.

2.1 Detailed Behavior Description

eSPEM provides a drop-in replacement package *ProcessBehavior*, that contains meta-classes and associations to reference *Behaviors* of the UML Superstructure from *WorkDefinitions* in SPEM. Figure 3 shows how this is done for *Activity*[1]. Other types of work in eSPEM (e.g., *TaskDefinition*) reference a *Behavior* in the same way.

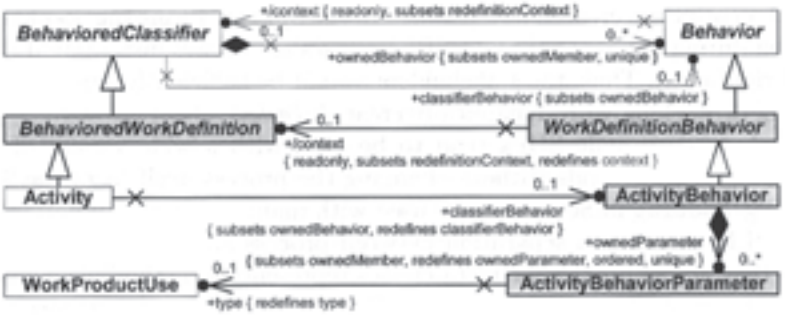

Fig. 3. *Activity* and associated *Behavior* in eSPEM

BehavioredWorkDefinition is the base class of all meta-classes that represent work. It inherits from *BehavioredClassifier* and *WorkDefinition* (not shown).

[1] A cross indicates a non-navigable association, and a dot indicates ownership of the association end by the opposite meta-class (UML 2.2 notation).

WorkDefinitionBehavior has the same structure and operational semantics as an *Activity* in UML.

The constructs presented in this section allow for an integration of the basic behavior modeling concepts from UML such as control and object flows, decisions, loops, and events for the different types of work in SPEM. However, we still need a way to express which *Roles* trigger events or make decisions.

2.2 Integration of Roles

Roles play an important part in SDP modeling. Figure 4 shows how eSPEM integrates roles into the behavior modeling concept. *Execution* is a *Relationship* that associates a *Role* for example with an *ActivityNode* (e.g., *DecisionNode* and *SendSignalAction*) and expresses that this *Node* is executed by the *Role*. Thus, it is now possible to specify that a decision is made or an asynchronous *Event* is triggered by a particular *Role*.

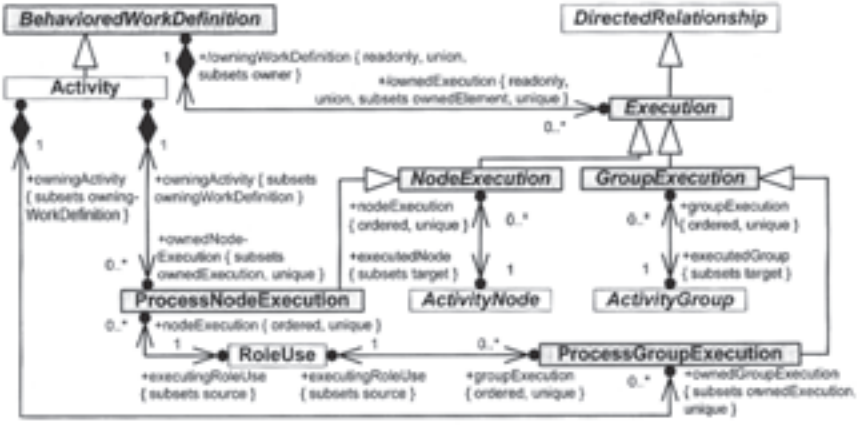

Fig. 4. *Roles* added to behavior modeling concept

Roles may also execute entire *ActivityGroups* to express that a *Role* executes all *Nodes* within that *ActivityGroup*. We added this concept because in UML an *ActivityPartition* (swimlane), which is an instantiable sub-class of *ActivityGroup*, provides no support for being executed by more than one *Element*. However, in SDPs there are often several roles executing the same work unit. This concept of eSPEM reduces the modeling effort and keeps the formalism easy-to-digest.

2.3 State Machines

We consider the use of *StateMachines* for *WorkProduct* life-cycle modeling to be a generic, detailed, and perfectly enactable approach. It outperforms concepts for *WorkProduct* life-cycle modeling found in other approaches (for example *WorkProductStatus* in SEMDM [14] that is a simple enumeration without formally

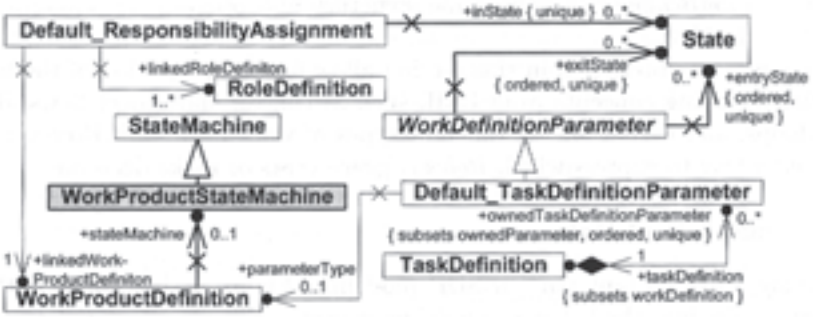

Fig. 5. *StateMachines* for *WorkProducts* in eSPEM

defined transitions). As mentioned above, interfacing *StateMachines* from SPEM is cumbersome and contradicts an easy-to-digest formalism. We integrate *StateMachines* for *WorkProduct* life-cycle modeling as shown in Fig. 5.

In eSPEM a *WorkProduct* may reference a specialized UML *StateMachine* that already groups a consistent and well-defined set of *States* and *Transitions*. With our concept for integrating *Roles* into the behavior model it is now also possible to specify which *Role* may trigger a *Transition* in the life-cycle model of a particular *WorkProduct* (e.g., who is allowed to approve a document). This supports automatic audit trails, one of our key requirements.

From SPEM we adopted the two association properties *entryState* and *exitState* of *WorkDefinitionParameter* that form the set of entry and exit states that are allowed. These constraints can be checked by a PEX and so increase the assessability of the enacted process.

Using the behavior model integration concept from eSPEM it is also possible to specify a life-cycle for all types of work in eSPEM (e.g., *TaskDefiniton*) using *StateMachines*. In practice, the life-cycle model is typically used for work that is not further decomposed (*TaskDefiniton*), is usually rather simple (e.g., created, running, or finished), and administered by a PEX or a dedicated bug tracking tool when the SDP is enacted.

2.4 Scheduling of Dynamically Created Tasks

Software development is often a creative process in which not all details are known a priori. This includes tasks that are usually created and planned while the process is already enacted. Although it would be possible to remodel the SDP whenever tasks are planned or new tasks are added, this approach suffers from the fact that the process description is no longer reusable across projects. We believe that an SDP model should reflect that tasks are dynamically created, planned, and executed at some potentially different point during an SDP. Due to its focus, SPEM neither provides support to define rules for planning tasks nor does it distinguish between the point of creation and the point of execution of tasks in a process.

Fig. 6. *TaskScheduler* and *ExecuteTaskDescriptorAction* in eSPEM

To support this, we added the meta-class *TaskScheduler* in eSPEM. A *TaskScheduler* is responsible for planning *Tasks* that may be dynamically created during the process.

In eSPEM an *Activity* may have a *TaskScheduler* associated that is responsible for scheduling the *Activities* and *TaskUses* within the WBS of the *Activity*, as shown in Fig. 6. To be compatible with the process, the *TaskScheduler* of an *Activity* must respect the precedence that is given by *WorkSequence* relationships and the behavior model of the *Activity*.

TaskSchedulers can also be used in the behavior model. An *ExecuteTaskDescriptorAction* takes the union of all sets of *Tasks* that arrive at its *TaskDescriptorInputPins* and delegates their scheduling to an associated *TaskScheduler*. The control flow returns from the *ExecuteTaskDescriptorAction* either when all *Tasks* are performed or when the *TaskScheduler* determines the end of the invocation. This concept allows for a separation between the instantiation and execution of *Tasks* using *ObjectFlows* to denote their "way" through the process.

With the concept presented in this section we added the ability of rule based ordering of *Tasks*, and a separation between instantiation and execution of *Tasks* in eSPEM.

2.5 Configuration of Processes

As outlined in Sect. 1, a significant effort has to be taken, when the target of a proxy in a process shall be changed. This is a quite common use case, e.g., when several suitable methods might be used for executing a task within a process. Choosing from these methods should be possible during process execution without enforcing any remodeling, tailoring or changing the underlying process model.

To reduce the modeling effort, we added the meta-class *ProcessToMethodMapping* in eSPEM, as shown in Fig. 7. A *ProcessToMethodMapping* expresses a possible mapping between a *MethodContentUse* and a corresponding *MethodContentElement*. Instantiable sub-classes of *ProcessToMethodMapping* exist for *Tasks* (see Fig. 7), *Roles*, *WorkProducts*, and *TaskSchedulers*. *ProcessToMethodMappings*

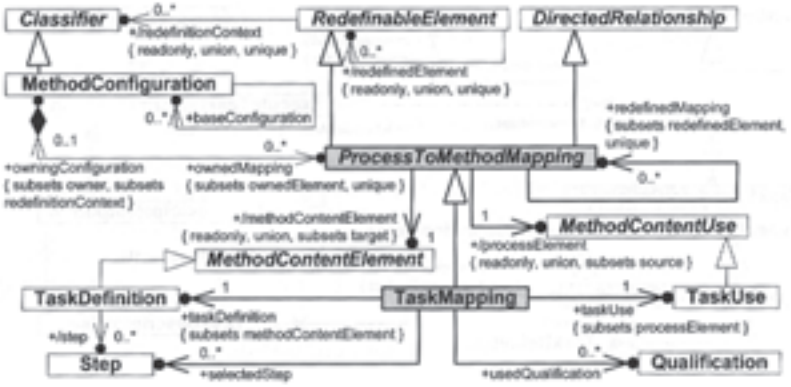

Fig. 7. *ProcessToMethodMapping* and *TaskMapping* in eSPEM

are composed into a *MethodConfiguration* that reflects one valid set of mappings between *MethodContentUses* and *MethodContentElements*. As a result only one proxy with its relationships must be modeled within a process for every possible configuration. Furthermore, the proxy needs no knowledge about its actual implementation because the mapping is separated from the proxy. This results in another benefit of our solution: The integration of new methods with existing processes is easier because the process does not need to be altered, which in turn eases evolution of already executed processes. Compatibility between the proxy and its implementation is ensured by additional OCL constraints we added in eSPEM. However, these are not discussed in this paper due to space restrictions.

2.6 Tool Support

In addition to the extension of SPEM itself we also implemented eSPEMs abstract syntax using the Eclipse Modeling Framework (EMF) [15]. This already enables precise modeling of SDPs. Furthermore, we implemented eSPEMs concrete syntax for *MethodContentElements*, *WorkDefinitionBehaviors* and *Actions*, and *StateMachines* using the Graphical Modeling Framework (GMF) [16] as diagram editors that work with our abstract syntax implementation of eSPEM. Both implementations are integrated into a PME that is used to model eSPEM-based SDPs [17]. Our PME also reuses parts of the Eclipse Process Framework (EPF) [18], which aims to be a PME for SPEM 2.0. Using our PME, we modeled two SDPs (Scrum and OpenUP) used in industry to test the usability of the constructs we added to eSPEM.

3 eSPEM Applied to SDPs

This section provides examples of how to use eSPEM to model aspects of SDPs that cannot be modeled with SPEM. In all these examples we use a mixture of standard UML and eSPEM notation.

3.1 Modeling the Behavior of SDPs

Figure 8 gives an overview of the Scrum behavior modeled with eSPEM (some elements are omitted). The `Product Owner` decides at the beginning of each `Sprint` whether to release the product or not. Depending on his decision either a `Development Sprint` or `Release Sprint` has to be performed next. With eSPEM we model this by means of a *DecisionNode*. A PEX that enacts this SDP model is now able to ask a person playing the role of the `Product Owner` what type of `Sprint` shall be performed and – depending on the answer – choose the right sub-process to guide and support the `Team`. Other SDPs require decisions as well, e.g., decisions that have to be taken when new risks are identified in the OpenUP or the V-Modell.

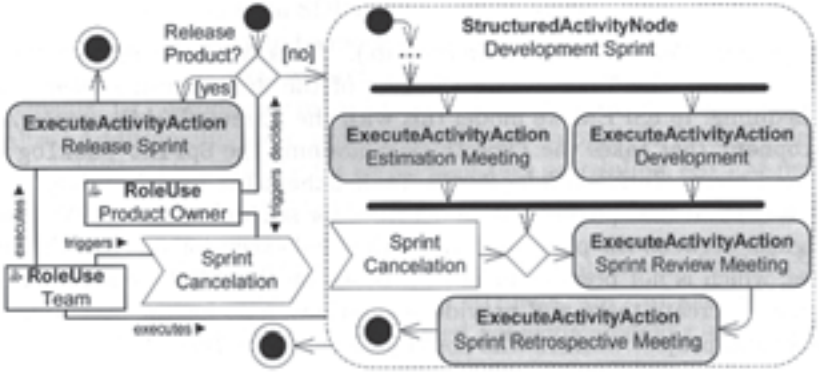

Fig. 8. Scrum behavior model with different *Executions*

Asynchronous *Events* are required by many SDPs. Typical examples are: 1) adding a new item to the `Risk List` (when using OpenUP or the V-Modell), 2) Changing `System-Wide Requirements` (when using the OpenUP), or 3) cancellation of a `Sprint` by the `Product Owner` or the `Team` (when using Scrum). With eSPEM this can be modeled using a *Send-/ReceiveSignalAction* which the corresponding *RoleUse* (e.g., `Product Owner`) `triggers` (see Fig. 8). A PEX can use this information to provide for example a button in a GUI to trigger this signal and execute the behavior to cancel the sprint.

3.2 Scheduler in SDP Behavior Models

Many SDPs require that *Tasks* are created and planned during enactment, e.g., *Tasks* on the `Work Items List` of the OpenUP and *Tasks* defined in a `Work Order` of the V-Modell. Scrum also requires planning of *Tasks*, e.g., in the course of the `Sprint Planning Meeting` *Tasks* are created, prioritized, and planned using the `Sprint Backlog`. Figure 9 shows how we can model this example using eSPEM. The `Sprint Backlog` is an output *Parameter* of the `Sprint Planning Meeting`. The character `C` within the *Pin* indicates that it is created during

Fig. 9. Scrum behavior model with *Scheduler*

that meeting (*ParameterEffectKind::create*). *Tasks* from the `Sprint Backlog` are executed by the `Team` during the rest of the `Sprint` using their priority for scheduling. In eSPEM we model this with the *ExecuteTaskDescriptorAction* `Development` that takes the *ObjectFlow* containing the `Sprint Backlog` as an input and uses a *TaskScheduler* `Scrum Task Scheduler` for scheduling.

With eSPEM it is possible to model rules for scheduling *Tasks*. We can also distinguish between the point of instantiation and execution of *Tasks* during the process, which is not possible in SPEM. Based on this SDP model, a PEX can dynamically create tasks and provide suggestions, e.g., for the execution order of *Tasks* and project workers who could perform these *Tasks*.

3.3 Configuring an SDP

SPEM distinguishes between process and methods. In order to get a complete SDP, methods have to be integrated with the process using a configuration. Figure 10 shows parts of two possible configurations for the *TaskUse* `Integrate and Create Build` from the OpenUP. The configuration `Default OpenUP` provides a basic build management setup (*TaskDefinition* `Integrate and Create Build`) using the contained mapping. The configuration `OpenUP with CI` provides extra steps to setup an continuous integration (CI) build (*TaskDefinition* `Setup CI Build`). Other SDPs require our enhanced configuration support as well, e.g., to configure the `Process Modules` of the V-Modell®XT for the different project type variants (e.g., `Project (Acquirer) with One Supplier` and `Project (Acquirer) with Several Suppliers`). eSPEM's configuration support can also be used when modeling Scrum, e.g., to define different types of scheduling strategies for *Tasks* executed during `Development` (e.g., based on the priority of *Tasks* or to optimize workload of the project workers).

4 Related Work

Since Osterweil's original approach of process programming [19], many PMLs have been proposed. Acuña and Ferré [4] discuss several of these PMLs and

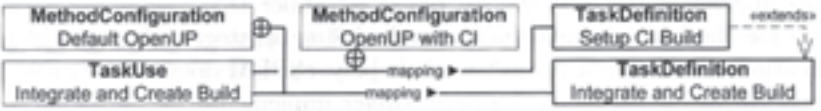

Fig. 10. Different OpenUP configurations

corresponding tools. Gruhn [3] demonstrates why the approaches were not accepted in industry and derives requirements for successful SDP modeling and execution environments. These requirements are addressed by our approach.

Standards like BPMN [11] or WS-BPEL [20] and its extension for People [21] were created to model and enact business processes. Although these approaches provide a reasonable behavior modeling and enactment concept, they do not provide other constructs from SDPs, e.g., roles, guidelines, responsibility assignments, and tools, that have to be modeled by means of BPEL variables or cannot be modeled at all. However, without these constructs a PEX cannot fulfill our requirements, i.e., a comprehensive EPG or integration of existing tools is hard to realize. Additionally, these approaches do not support a formally defined, fine-grained life-cycle modeling for artifacts or task scheduling. eSPEM provides a tightly integrated, fine-grained behavior modeling approach that supports the mentioned constructs from SDPs.

Bendraou et al. [22] present an extension of the SPEM standard called xSPEM and focus on SDP validation using timed Petri nets. xSPEM also adds *Events* for SPEM-*Activities* but lacks a fine-grained behavior modeling approach with decisions, life-cycles as well as task scheduling.

Seidita et al. [23] extend SPEM to support the modeling of agent oriented methodologies [24] but do not define a fine-grained behavior modeling concept.

Both extensions do not address the incomplete configuration support of SPEM, as we do.

Another meta-model driven approach for describing development methodologies is the ISO/IEC standard SEMDM [14]. It does not use OMG's strict meta-modeling approach but uses the power type pattern [25]. This pattern was adopted for meta-modeling in the domain of software development methodologies by Henderson-Sellers and Gonzalez-Perez [26]. The rationale for using the power type pattern is to be able to define instance features within the meta-model, which is not supported when using strict meta-modeling. However, SEMDM does not provide a standardized notation, which clearly contradicts an easy-to-digest formalism. Additionally, SEMDM lacks a fine-grained behavior modeling concept. Similar to [27,28], we use *Kinds* in a model library (not in the meta-model) to specify common instance properties for a set of model elements.

The concept of rule-based task execution for SDPs has been studied before. Heimann et al. [29] present DYNAMITE, which is based on instance level task nets. DYNAMITE uses PROGRES [30] for rule-based transformations of whole subgraphs within task nets. This already allows for a basic scheduling of tasks. However, DYNAMITE does neither support modeling additional properties for tasks nor does it support modeling roles. Thus, some scheduling strategies are

hard to implement, e.g., a priority based scheduler that requires a property *priority* for the tasks it schedules, or scheduling strategies that consider the qualifications of roles that project workers play. eSPEM does support roles and instance features and therefore *TaskScheduler* implementations are able to use the additional information to get more accurate scheduling results.

Using UML for modeling SDPs is a common approach. Bendraou et al. [9] compare six UML-based languages for modeling SDPs including SPEM 1.1, SPEM 2.0 [1], UML4SPM [31], and other approaches [32,33,34]. We will give a short comparison of these approaches with eSPEM in the following.

Closest to our work is UML4SPM [31]. UML4SPM is based on SPEM 1.1 and UML 2.0 behavior modeling concepts. In [35] UML4SPM is mapped to WS-BPEL for enactment support. As mentioned, WS-BPEL cannot fulfill our requirements. A more recent approach to gain enactment support for UML4SPM is shown in [36]. In this paper the authors present an execution model based on the OMG proposal for an executable UML subset [37], as well as an implementation of the execution model for UML *Activity* and *Actions* using Kermeta [38]. This enables an execution and simulation of UML4SPM-based models. However, UML4SPM is based on SPEM 1.1 and therefore does not provide sophisticated tailoring and configuration support. Additionally, UML4SPM does not use *StateMachines* for life-cycle modeling.

Chou [33] uses a subset of UML 1.4 activity and class diagrams in combination with a proprietary object-oriented process programming language. The approach suffers from the fact, that code in the low-level programming language is not derived from the diagrams.

Di Nitto et al. [32] propose a UML 1.3-based framework to model SDPs. They do not extend the UML meta-model or use stereotypes. The framework elements, e.g., `SoftwareActivity`, are instances of the UML meta-class *Class*. However, using plain *Classes* is a major drawback of this approach because all process elements have the same notation and semantics.

Franch et al. [34] present PROMENADE, an extension of the UML 1.x meta-model, and add essential concepts for SDP modeling, i.e., roles, tasks, and documents. However, they do not add dedicated relationships, e.g., responsibility assignments that are available in eSPEM.

Engels et al. [39] show how the concepts in UML can be used for process modeling. However, essential concepts of SDP modeling are missing in UML, e.g., work products and responsibility assignments. This results in an incomplete and imprecise SDP description.

Other approaches use UML and extensions through stereotypes for SDP modeling [40] or SPEM itself [1]. This allows to use standard UML modeling tools and the behavior modeling concepts from UML. However, several other problems arise. Stereotypes change the semantics of UML elements when being applied to them but have no influence on the language structure as defined by the UML meta-model. Consequently, "type-safety" and multiplicities of common relationships for SDP modeling (e.g., responsibility assignments and relationships between work products) must be re-implemented (e.g., as constraints

for the stereotypes). This contradicts our requirement of an easy-to-digest formalism and limits the support that a general purpose UML modeling tool can give when creating or editing model elements with stereotypes.

5 Conclusion and Future Work

Given our requirements, the current SPEM standard has a few issues that we have identified by modeling exemplary SDPs. With eSPEM we have provided an extension of SPEM that addresses the identified issues. eSPEM supports a fine-grained behavior and life-cycle modeling, definition of task scheduling strategies, and an enhanced configuration support. None of the approaches known from literature does support all of the features in eSPEM that we consider to be required for precise modeling and reasonable enactment support of SDPs.

Our future work will focus on an enhanced tool support for eSPEM. This includes a full implementation of eSPEM's concrete syntax and additional tooling to improve usability of our PME. In addition to that, we will also work on implementing a PEX for eSPEM-based SDP models. Research in this field will include the formal definition and implementation of the operational semantics of eSPEM's behavior model, integration of existing tools and their data formats, traceability support for artifacts, and process evolution. Providing the combination of a PME and a PEX will also allow to empirically evaluate the impact of computer-aided process enactment on real development projects, and an adaption of eSPEM by practitioners.

References

1. Object Management Group: Software & Systems Process Engineering Meta-Model Specification Version 2.0 (April 2008)
2. Object Management Group: Unified Modeling Language: Superstructure Version 2.2 (February 2009)
3. Gruhn, V.: Process Centered Software Engineering Environments – A Brief History and Future Challenges. Annals of Software Engineering 14(1-4), 363–382 (2002)
4. Acuña, S.T., Ferré, X.: Software Process Modelling. In: Proc. World Multiconf. on Systemics, Cybernetics and Informatics, Orlando, FL, pp. 237–242 (2001)
5. Conradi, R., Jaccheri, M.L.: Process Modelling Languages. In: Derniame, J.-C., Kaba, B.A., Wastell, D. (eds.) Promoter-2 1998. LNCS, vol. 1500, pp. 27–52. Springer, Heidelberg (1999)
6. Jaccheri, M.L., Baldi, M., Divitini, M.: Evaluating the requirements for software process modeling languages and systems. In: Process support for Distributed Team-based Software Development (PDTSD 1999), Orlando, FL, pp. 570–578 (1999)
7. CMMI Product Team: CMMI for Development Version 1.2. Carnegie Mellon University – Software Engineering Institute, Pittsburgh, PA (August 2006)
8. ISO/IEC: ISO/IEC 15504-5:2006 – Information technology – Process Assessment – Part 5: An exemplar Process Assessment Model (March 2006)
9. Bendraou, R., Jezequel, J.M., Gervais, M.P., Blanc, X.: A Comparison of Six UML-Based Languages for Software Process Modeling. IEEE Transactions on Software Engineering 99 (2009) (PrePrint)

32. Di Nitto, E., Lavazza, L., Schiavoni, M., Tracanella, E., Trombetta, M.: Deriving executable process descriptions from UML. In: Proc. 24th Intl. Conf. on Software Engineering, New York, NY, pp. 155–165 (2002)
33. Chou, S.C.: A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language. J. of Object Technology 1(4), 137–163 (2002)
34. Franch, X., Ribó, J.M.: A Structured Approach to Software Process Modelling. In: Proc. 24th EUROMICRO Conf., Washington, DC, vol. 2, pp. 753–762 (1998)
35. Bendraou, R., Sadovykh, A., Gervais, M.P., Blanc, X.: Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach. In: Proc. 33rd EUROMICRO Conf. on Software Engineering and Advanced Applications, Lübeck, Germany, pp. 314–321 (2007)
36. Bendraou, R., Jezéquél, J.M., Fleurey, F.: Combining Aspect and Model-Driven Engineering Approaches for Software Process Modeling and Execution. In: Wang, Q., Garousi, V., Madachy, R., Pfahl, D. (eds.) ICSP 2009. LNCS, vol. 5543, pp. 148–160. Springer, Heidelberg (2009)
37. Object Management Group: Semantics of a Foundational Subset for Executable UML Models RFP (April 2005)
38. Muller, P.A., Fleurey, F., Jezéquél, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
39. Engels, G., Förster, A., Heckel, R., Thöne, S.: Process Modeling using UML. In: Dumas, M., van der Aalst, W., ter Hofstede, A. (eds.) Process-Aware Information Systems, pp. 85–117. John Wiley & Sons, Chichester (2005)
40. Jäger, D., Schleicher, A., Westfechtel, B.: Using UML for Software Process Modeling. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 91–108. Springer, Heidelberg (1999)