

Chapter 4

Process Modeling Notations and Tools

This chapter introduces notations for process modeling and gives an overview of tool support for process modeling and management. The chapter is structured into three main parts. First, it introduces a set of criteria for process modeling notations in order to enable the reader to distinguish different process modeling notations and to understand that different purposes might be addressed by different notations. Second, it discusses two different process modeling notations, namely, MVP-L and SPEM 2.0, and characterizes them according to the previously defined criteria. Finally, it introduces process management tools by discussing the ECMA/NIST framework and the Eclipse Process Framework (EPF) Composer. Figure 4.1 displays an overview of the chapter structure.

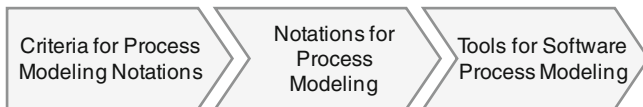


Fig. 4.1 Chapter structure

4.1 Objectives of This Chapter

After reading this chapter, you should be able to

- Distinguish different process modeling notations and assess their suitability with respect to different purposes
- Explain and use the basic concepts of MVP-L
- Explain and use the basic concepts of SPEM 2.0
- Understand and explain the components of process management tools

4.2 Introduction

When we think of process modeling notations, we can identify a plethora of different approaches. This is due to the fact that during the historical development of process modeling notations, different communities have influenced the discipline of process modeling. In terms of software engineering processes, two major groups that influenced the development of process modeling notations can be identified [1].

The first group was significantly influenced by tool developers and programmers. Within this group, notations for the representation of processes were developed or adopted aimed at creating representations that could be interpreted by machines. Thus, this group focused on process automation and the notations used were typically not designed to be interpreted by humans. The underlying vision was to create software development environments where the execution of software development tools would be controlled by a process-driven engine. The main focus was on small, low-level processes such as the code–compile–test–fix cycle. As a result, this approach focused on processes with a high potential of automation.

The second group has its origins in the community that was concerned with software process improvement. In this discipline, the aim was to make software development more mature by means of introducing best practices and establishing learning cycles. For this reason, the need arose to represent software processes in order to understand and improve the processes of software development performed by humans. The notation constructs developed in this context aimed at describing real-world concepts and creating models that humans can interpret. This approach, and in particular the representation of software engineering processes, focused on higher level processes and a minor degree of automation. Therefore, processes are described in a more informal and less detailed way and, most importantly, they provide guidance that can be interpreted and enacted by humans. In this context, process guides based on natural notation became popular. They concentrate on providing people with the information necessary to appropriately enact the process.

Currently, an abundance of different process modeling notations exists and, therefore, a strong need for standardization has developed. As a result of this development, the Software Process Engineering Metamodel (SPEM) was created. Its goal is to enable the representation of different software engineering concepts.

4.3 Criteria for Assessing Process Modeling Notations

The multitude of existing process modeling notations has been developed due to different motivations and needs. As needs usually differ greatly for different stakeholders, purposes, and contexts, there is no best representation for processes, and thus different representations cannot be assessed from a general point of view. But it can be useful to compare different concepts in order to understand the specific aspects that are addressed by a specific representation.

This section will introduce concepts for characterizing process modeling notations and furthermore define requirements for process modeling notations from different perspectives. These concepts are useful for comparing different notations for the representation of processes.

4.3.1 Characteristics of Process Modeling Notations

In order to understand the context and motivation of a certain representation, Rombach and Verlage [1] use the following aspects for characterizing process modeling notations.

4.3.1.1 Process Programming vs. Process Improvement

A major distinction can be made between process modeling notations for the implementation of processes (i.e., process programming) and notations for the conceptual modeling of processes (i.e., process improvement). Process programming notations focus on a representation for interpretation and execution by machines. Process improvement notations focus on representation of real-world concepts and provision of a representation that can be interpreted by humans.

4.3.1.2 Hidden vs. Guiding

When the process model is used, the representation of the process models can be hidden or presented to the process user. When hidden, the process instantiation is completely encoded in the process models or tools that support process enactment. Thus, only filtered information is provided concerning the current project state. If used for guiding, the process models themselves are used to inform the user and to provide guidance during process instantiation.

4.3.1.3 Prescriptive vs. Proscriptive

In the early days of software process research, the main focus was placed on automating process execution with the help of software development tools. Therefore, the user of such tools would be guided by an execution mechanism in a prescriptive manner. This approach of prescribing the process and thus also the human activities has been subject to criticism and is difficult to implement. The proscriptive approach represents a nonrestrictive way of formulating processes. The process models provide guidance in order to enable performance of the required process steps, but process users have a certain freedom in deciding which actions to take at a particular stage of the project.

4.3.1.4 Single Person vs. Multiperson

Software development projects are not performed by a single person and, in consequence, collaboration and cooperation between persons, teams, and organizations is highly relevant. Process models should support all these different levels in order to make collaboration and cooperation possible. Historically, process representations have evolved from a single-person focus in order to ensure proper application of specific techniques by individuals. For the purpose of cooperation, a multiperson focus is needed in order to coordinate the processes of different persons. Therefore, a process representation should contain constructs for modeling concepts of collaboration.

4.3.2 Requirements for Process Modeling Notations

In the following, a set of requirements for process modeling notations will be described in accordance with [1]. The fulfillment of these requirements can be seen as an indicator for the suitability of the notation to support process management for software engineering organizations. Based on the viewpoint, the purpose, and the context, different requirements might be relevant. A process engineer who wants to automate a build process of a business unit might select different requirements than an education department that aims at introducing a company-wide training program. The stated requirements help to find suitable process modeling notations by first selecting the relevant requirements and afterwards selecting such notations that fulfill the requirements. The following requirements can be applied [1].

- *R1—Natural Representation:* A process modeling notation should not only be able to capture all relevant aspects of software development, but it should also be able to represent these aspects in a natural, intuitive, and easy-to-identify manner. A mapping between real-world phenomena and process model elements that is as complete as possible facilitates the modeling and maintenance of these models.
- *R2—Support of Measurement:* A process modeling notation should take into account the measurability of the process model. In order to enable software process improvement, the impact of different technologies on products and processes has to be observed. Furthermore, the scientific evaluation of the efficiency and effectiveness of these technologies should be based on measurement. For this reason, the notation has to take into account the definition of attributes and measurement within process models.
- *R3—Tailorability of Models:* On the one hand, a process modeling notation should enable a generic representation of information in order to allow for process models that can describe commonalities of processes from several different projects. On the other hand, no development project is completely similar to another one and therefore, the process environment is most likely to

change for each project. Thus, in planning a project, the differences must be considered and the process model has to be instantiated and tailored accordingly. The use of tailorable models limits the number of process models and thus reduces maintenance efforts. Therefore, concepts for defining and supporting process variability and tailoring are needed.

- *R4—Formality*: A process modeling notation should allow for the creation of process models with a certain degree of formality. Formality is needed to support communication among different process participants and to foster a common understanding of the process model by different people. Fulfillment of this requirement means that process model constructs are defined formally within the process model.
- *R5—Understandability*: Understandability is a key aspect of a process modeling notation, as process models are used as a reference during projects. Most activities related to process engineering rely on human interpretation rather than interpretation by a machine and understandability is therefore a crucial factor for the success of any process representation. Understandability refers to the style of presentation and to how difficult it is for its users to retrieve needed information.
- *R6—Executability*: A process modeling notation should support the interpretation and execution of the process representation by a machine. This need arises due to the fact that standard procedures of software development are often supported by tools that aim at providing automated support for the process user.
- *R7—Flexibility*: A notation for process representation should account for handling decisions made by humans during process performance. These decisions are characterized by creativity and nondeterminism. A process modeling notation thus should contain constructs that are capable of capturing these aspects.
- *R8—Traceability*: Traceability should be ensured within and across layers of abstraction (i.e., horizontal and vertical traceability). This means that, for each piece of information, it should be possible to determine its context, the processes that rely on it, and how it was transformed. A process modeling notation should thus support process representations that provide constructs for the explicit description of different relationships between various process elements.

These characteristics and requirements can be used to define a framework that helps to distinguish different process modeling notations and their purpose. All elements of this framework are summarized in Table 4.1 (adapted from [1]). For the evaluation of requirements satisfaction, (+) represents full, (O) partial, and (–) no fulfillment of the respective requirement.

In the following sections, two software process modeling notations, MVP-L and SPEM 2.0, will be introduced. MVP-L represents a notation that offers a comprehensive set of modeling constructs. SPEM 2.0 will be introduced because it has the potential to become a future process model notation standard. The framework of characteristics and requirements that was introduced earlier will be used to give an overview and characterization of these notations.

Table 4.1 Characterization framework

Characterization	
Process programming vs. improvement	Prescriptive vs. proscriptive
Hidden vs. guidance	Single person vs. multiperson
Requirements satisfaction	
R1—Natural representation	(+/O/–)
R2—Support of measurement	(+/O/–)
R3—Tailorability of models	(+/O/–)
R4—Formality	(+/O/–)
R5—Understandability	(+/O/–)
R6—Executability	(+/O/–)
R7—Flexibility	(+/O/–)
R8—Traceability	(+/O/–)

4.4 Multi-view Process Modeling Language

4.4.1 Overview

Multi-view process modeling language (MVP-L) was developed in the 1980s at the University of Maryland. Subsequent development was conducted at the University of Kaiserslautern, Germany. MVP-L has its origins in the Multi-view process modeling (MVP) project, which focused on process models, their representation, and their modularization according to views, as well as their use in the context of software process improvement, namely, the quality improvement paradigm. MVP-L was developed to support the creation of descriptive process models, packaging of these models for reuse, integration of the models into prescriptive project plans, analysis of project plans, and use of these project plans to guide future projects [2].

The main focus of MVP-L is on modeling “in-the-large.” It is assumed that the ability to understand, guide, and support the interaction between processes is more beneficial than the complete automation of low-level process steps [2].

4.4.2 Concepts

The main elements that are used in MVP-L for the description of process models are processes, products, resources, and quality attributes, as well as their instantiation in project plans [2]. A process model is actually a type description that captures the properties common to a class of processes. For easy adaptation of process models to different project contexts, the process models are structured using the concepts of a process model-interface and a process model-body. An interface describes a generalization of the formal parameters that are relevant to all models of a particular kind. As an example, a process model “Design” (Fig. 4.2, based on [2])

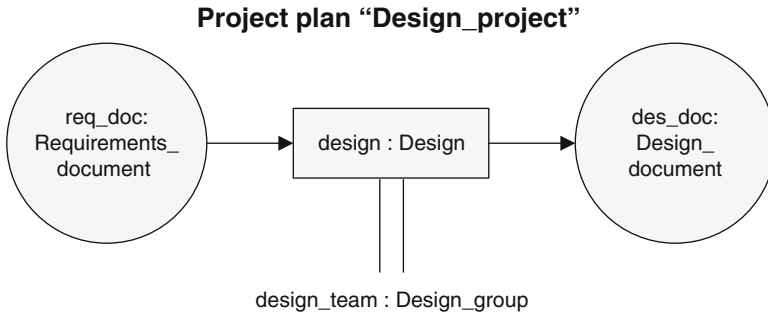


Fig. 4.2 Example of process model “Design”

could describe a class of processes that require an input of the product type “Requirements_document,” which must produce an output of the product type “Design_document,” and which must be executed by a resource of the type “Design_group.” These product and resource model declarations are part of the interface of the process model “Design.” The actual implementation of the process model is “hidden” in the body of the process model. Thus, MVP-L models implement the important concept of information hiding [3]. The model-body contains information that is only visible internally, whereas the model-interface contains information that is visible to other models. By implementing the concept of information hiding, changes to models or parts of models can be performed and handled locally without affecting other models.

4.4.3 Notation Constructs

Processes, products, and resources can be used for modeling the basic elements of a software project. Attributes can be used for defining specific properties of these three basic elements. MVP-L calls the constructs for describing these elements “models.” However, they can be understood as types [2].

- *Product_model*: Software products are the results of processes for development or maintenance. In addition to the final software product, by-products, artifacts, and parts of a product’s documentation are called products as well.
- *Resource_model*: Resources are the entities that are necessary for performing the processes (e.g., people or tools).
- *Process_model*: Processes are the activities that are performed during a project. They produce, consume, or modify products.
- *Attribute_model*: Attributes define properties of products, resources, and processes. The attributes that are used are process_attribute_model, product_attribute_model, and resource_attribute_model. Attributes correspond to measures and their values correspond to specific measurement data.

In the following, these constructs will be discussed in more detail and examples will be given for illustration purposes. The following descriptions and examples are based on the MVP-L language report [2].

4.4.3.1 Product Models

Product models describe the structure and properties of a class of software products. Product models do not only describe code artifacts, but all artifacts that are part of software development activities and supporting activities. Each product representation consists of an interface and a body. Information in the `<product_interface>` is visible to other objects. The product attributes are declared in the `<exports>` clause, and their type must first be imported in the product interface's `<import>` clause.

```
Example—Product Model:Requirements document

product_model Requirements_document (status_0 : Product_status) is
  product_interface
    imports
      product_attribute_model Product_status;
    exports
      status : Product_status := status_0;
  end product_interface

  product_body
    implementation
      {textual description}
    end product_body
end product_model Requirements_document.
```

The product model “Requirements_document” imports a product attribute model “Product_status” in order to declare a product attribute “status.” The formal instantiation parameter “status_0” is used to provide the initial value for the attribute.

4.4.3.2 Resource Models

Resource models describe resources involved in performing a process. Resources can be differentiated into organizational entities (e.g., groups or teams) and human individuals (active resources) or tools (passive resources). Active resources perform processes and passive resources support the performance of processes. Note that traditional software tools can be represented in MVP-L as resources as well as processes. A compiler, for example, could be represented as an MVP-L process

integrated into an MVP project plan dealing with program development. In contrast, an editor may be used as a passive resource within a project plan to support the design process. Like product models, resource models consist of a <resource_interface> and a <resource_body>. For instantiation, parameters can be defined. Parameters are special kinds of attributes for passing values to objects when the objects are instantiated. In the example below, the parameter “eff_0” of the type “Resource_effort” is used. It contains the effort that is available to a designer for the execution of the process in the context of a specific project plan.

Example – Resource Model: Designer

```

resource_model Designer(eff_0: Resource_effort) is
  resource_interface
    imports
      resource_attribute_model Resource_effort;
    exports
      effort: Resource_effort := eff_0;
  end resource_interface

  resource_body
    implementation
      { - An instance of this model represents a single member
        of the design team.
        - Persons assuming the role of a designer must be qualified.}
    end resource_body
end resource_model Designer

```

4.4.3.3 Process Models

Process models contain the information that is relevant for performing a specific task. In particular, process models combine the basic elements of products and resources in a manner that allows producing the resulting product. Similar to product and resource models, process models are structured into a model-interface and a model-body.

The process interface is described through <imports>, <exports>, <consume_produce>, <context>, and <criteria> clauses, as shown in the following example, which describes an exemplary design process. The process body is defined in terms of an <implementation> clause. The <imports> clause lists all externally defined models used to declare formal parameters within the <product_flow> clause or attributes within the <exports> clause. The <exports> clause lists all externally visible attributes that can be used by other models. These constructs provide a clear

interface to other models. In the example described later, the attribute “effort” of the type “Process_effort” is made available to all models importing the process model “Design.” A *product flow* is implemented in the process model through the <product_flow> clause, which lists all products that are consumed, produced, or modified. Products that are modified are declared in the <consume_produce> clause. For the exemplary process model “Design,” a product “req_doc” of the type “Requirements_document” is consumed and a product “des_doc” of the type “Design_document” is produced.

Furthermore, *constraint-oriented control flows* can be defined by using *explicit entry and exit criteria* as well as *invariants* within the MVP-L process models. The <criteria> clause within the process model interface describes the pre- and postconditions that have to be fulfilled in order to enter or exit the respective process. In addition, invariants are used to describe states that need to be valid throughout the enactment of the process. Criteria are specified as Boolean expressions. The expression following the keyword <local_entry_criteria> defines the criteria necessary to execute the process in terms of locally defined attributes and local interface parameters. In this example, the local invariant specifies that the actual effort spent for any instance of the process model “Design” should never exceed a value specified by “max_effort.” Invariants can be used to implement elements that need to be tracked permanently during process performance and are not allowed to exceed a certain limit. In particular, this accounts for monotonously rising or falling elements. Project effort, for example, should not exceed its maximum value. In the example, the local entry criteria state that any process of the type “Design” can only be executed if the attribute “status” of the product “req_doc” has the value “complete” and the attribute “status” of the product “des_doc” has either the value “non_existing” or “incomplete.” The expression following the keyword <local_exit_criteria> defines the criteria expected upon completion of process execution in terms of local attributes and the local interface. In the example, the locally expected result upon completion is that the attribute “status” of the product “des_doc” has the value “complete.” Thus, the concept of entry and exit criteria can be used to describe an implicit constraint-oriented control flow. MVP-L also provides constructs for defining global criteria and invariants that address global attributes, such as calendar time.

The <implementation> clause describes how an elementary process is to be performed. This can either be a call of a supporting tool, or simply an informal comment characterizing the task at hand for performance by a human. Processes are related to products via explicit <product_flow> relationships, to attributes via <criteria> clauses, and to resources via a separate <process_resources> clause. In the example of the process model “Design,” a resource “des1” of the type “Designer” is designated to execute any process of the type “Design.”

Example – Process Model: Design

```

Process_model Design(eff_0: Process_effort, max_effort_0: Process_effort) is

process_interface
    imports
        process_attribute_model Process_effort;
        product_model Requirements_document, Design_document;
    exports
        effort: Process_effort := eff_0;
        max_effort: Process_effort := max_effort_0;
    product_flow
        consume
            req_doc: Requirements_document;
        produce
            des_doc: Design_document;
        consume_produce
    entry_exit_criteria
        local_entry_criteria
            (req_doc.status = "complete") and (des_doc.status =
            "non_existent" or des_doc.status = "incomplete");
        local_invariant
            effort <= max_effort;
        local_exit_criteria
            des_doc.status = "complete";
end process_interface

process_body
    implementation
        {textual description}
end process_body

process_resources
    personnel_assignment
        imports
            resource_model Designer;
        objects
            des1: Designer;
        tool_assignment
and process_resources
end process_model Design

```

4.4.3.4 Attribute Models

Each attribute model refers to a certain model type and consists mainly of a definition of the `<attribute_model_type>` (and `<attribute_manipulation>`, which is not discussed here). The `<attribute_model_type>` characterizes the type of values the attribute stores. This type could be an integer, a real, string, Boolean, or enumerated type (see example).

Example - Attribute Model: Product status

```
product_attribute_model Product_status () is
    attribute_type
        ( "non_existing", "incomplete", "complete" );
    ...
end product_attribute_model Product_status
```

4.4.4 Instantiation and Enactment

The basic MVP-L models described so far can be refined and combined to create complex process models, which can be used to describe typical software and systems engineering processes. The instantiation of a process model allows operationalizing the process model and creating a concrete project plan, which can then be used for project analysis or execution. This section introduces the MVP-L representation of project plans, with an emphasis on the instantiation of processes and process enactment as described in [2]. The creation of project plans in MVP-L allows for creating executable `<project_plan>` objects.

4.4.4.1 Instantiation

Software process models in MVP-L are instantiated through `<project plan>` objects. A `<project_plan>` is described through `<imports>`, `<objects>`, and `<plan_object_relations>` clauses. The imports clause lists all models that are used to specify the process, product, and resource objects that make up the project plan. These objects are declared in the `<objects>` clause. The objects are interconnected according to their formal interface definition in the `<plan_object_relations>` clause. A project plan needs to be interpreted by a process engine (a human or a computer) in order to enact the contained processes.

Example – Project Plan: Design Project 2

```

project_plan Design_project_2 is
    imports
        product_model Requirements_document, Design_document;
        process_model Design;
        resource_model Design_group;
    objects
        requirements_doc: Requirements_document(„complete“);
        design_doc: Design_document(„non_existent“);
        design: Design(0, 2000);
        design_team: Design_group(0);
    object_relations
        design(req_doc => requirements_doc, des_doc => design_doc,
            designers => design_team);
end project_plan Design_project_2

```

The project plan example consists of four objects: one process “design,” two products “requirements_doc” and “design_doc,” and one resource “design_team.” The interconnection of these products and the resource with the process “design” is performed according to the formal interface specification of the process model “Design.” In this example, a complete requirements document (“requirements_doc”) is provided, the design document “design_doc” does not yet exist, and the time that is available for the performance of the process “design” is restricted to 2000 time units. Finally, only members of the “Design_group” are allowed to perform the process “design.”

4.4.4.2 Enactment

The notion of a *project state* is the basis for the enactment model in MVP-L [2]. A project state is defined as the set of all attribute values (i.e., all attributes of all objects instantiated within a project plan). Thus, the project state provides valuable information about the status of the projects at any given time. This is an important foundation for effective project control. The initial project state is defined in terms of the initial values of all user-defined attributes and the derived values of built-in attributes.

The values of attributes of the built-in type “Process_status” depend on the entry and exit criteria. The only triggers that change the current project state are user invocations of the kind “start(<object_id>)” and “complete(<object_id>)” to start and complete processes, or the invocation “set(. . .)” to address external changes of attributes. In each case, the new values of all user-defined and built-in attributes

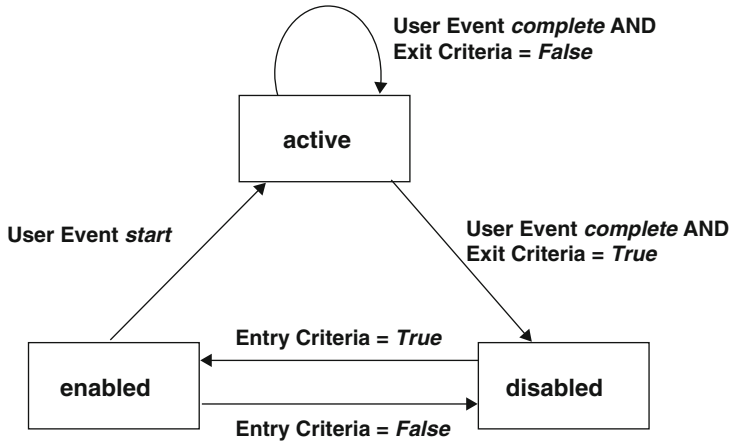


Fig. 4.3 State transition model for processes

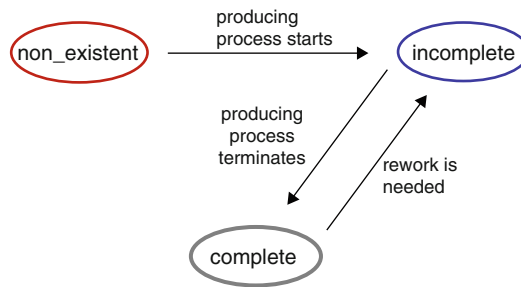


Fig. 4.4 State transition model for products

are computed to determine the new project state. A new project state provides information about the processes that are in execution (i.e., the value of the process status is “active”), ready for execution (i.e., the value of the process status is “enabled”), or not ready for execution (i.e., the value of the process status is “disabled”). The different states of a process can be represented in a state transition model (Fig. 4.3). Starting in the disabled state, processes may only get enabled when the entry criteria are true. An enabled process may get active when it is triggered by a user with the “start” invocation. As long as the exit criteria are not fulfilled and the user does not trigger the user invocation “complete,” the process will remain in the active state. When the exit criteria are fulfilled and the user invocation “complete” is triggered, then the process gets disabled. Additionally, for each project state, the state of the associated work products is represented as “non_existent,” “incomplete,” or “complete” with the built-in type “Product_status.”

Consequently, a state transition model can also be defined for products (Fig. 4.4). At the beginning, the product does not exist. When the producing process starts, the product state changes to incomplete. Finally, when the producing process

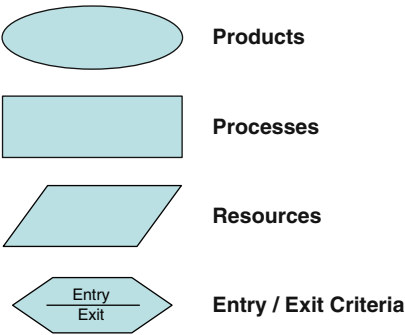


Fig. 4.5 Elements of graphical MVP-L representation

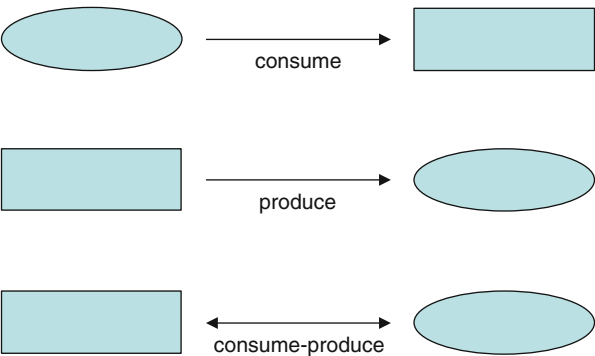


Fig. 4.6 Elements of MVP-L product–process relations

terminates, the product state turns to complete. When rework is needed, several iterations between the product states complete and incomplete are possible.

In addition to the textual representation of MVP-L, a graphical representation is defined for MVP-L in order to facilitate understanding and support process model reviews by process users [4]. Figure 4.5 introduces a graphical representation for MVP-L’s products, processes, resources, and entry as well as exit criteria. Figure 4.6 displays the product–process relationships.

For illustration purposes, a simple example of an actual project is provided (Fig. 4.7). This example illustrates the notion of the project state as well as the capabilities of MVP-L in implementing a constraint-oriented control flow using entry and exit criteria. The exemplary process consists of three process instances, namely, requirements specification, design, and coding. In this example, the process is strictly sequential. There are four work products that constitute the product flow within this process. According to Fig. 4.6, an arrow from a product to a process indicates that a product is consumed by this process. An arrow pointing from a process to a product indicates that a product is produced by this process. Control of the process flow is realized implicitly via pre- and postconditions of the process. Since the process is sequential in our case and every subprocess creates one work product, the entry

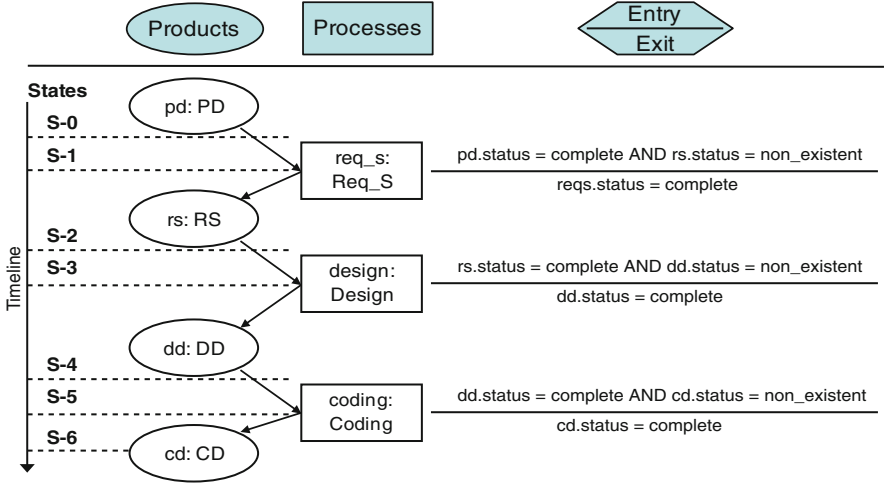


Fig. 4.7 Exemplary process in MVP-L graphical representation

condition could be described as follows: *The prior work product has to be complete AND the next work product has to be nonexistent.* The exit condition is defined as: *The next work product has been completed.* In the right column of Fig. 4.7, entry and exit criteria are explicitly specified. For example, in order to begin coding, the status of the design document “dd” has to be “complete” and the status of the code document “cd” has to be “non_existent.” In order to finish coding and to exit the process, the status of the code document has to be “complete.”

Finally, on the left of Fig. 4.7, project states are represented that correspond to the enactment scenario provided in the state table in Fig. 4.8 (adapted from [5]). The state table provides a sequence of project plan execution states. Starting in project state S-0, let us assume that the product description “pd” is already “complete” and other products are “nonexistent.” As the product description is “complete,” the process instance requirements specification can be enabled. The process instance is initiated with the invocation “start(req_s)” and state S-1 is reached. In S-1, the requirements specification process instance is “active” and the requirements specification document “rs” is being produced and is therefore in the state “incomplete.” Upon completion of the requirements specification, “complete(req_s)” triggers another project state change. In state S-2, the requirements specification document is “complete,” and thus the exit criterion for requirements specification is fulfilled. The requirements specification process instance gets “disabled.” Now the entry conditions for the design process are fulfilled, state S-3 can be achieved (“start(design)”), and the design process instance becomes “active.” The active design process instance creates the design document and therefore the design document is “incomplete.” All other process instances are “disabled.” State S-4 is triggered upon completion of the design document (i.e., its exit criterion is fulfilled and “complete(design)” is triggered). Now the entry criteria for the coding process are fulfilled and state S-5 can be entered.

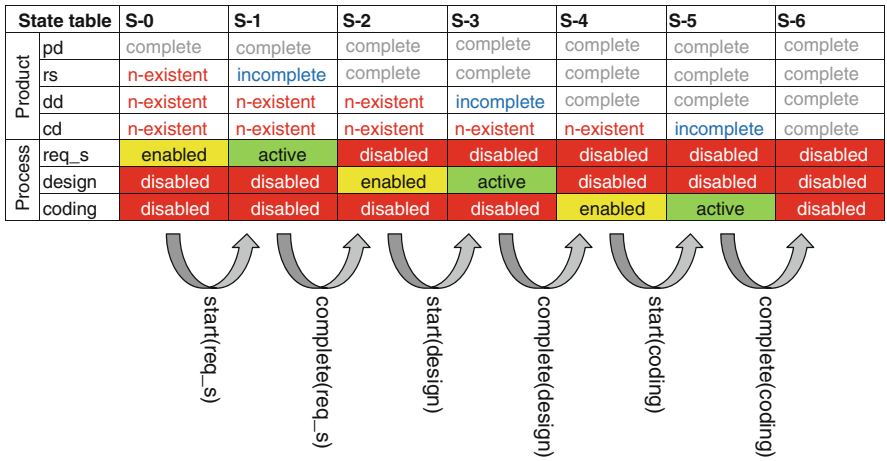


Fig. 4.8 Example of a state table

In S-5, the code document is under creation (code document: “incomplete”) and the coding process instance is “active.” When the code document reaches the state “complete,” the exit criterion for coding is fulfilled and state S-6 is reached through user invocation “complete(coding).” In S-6, all work products are “complete” and all process instances are “disabled” (Fig. 4.8, adapted from [5]).

In this section, the basic concepts of MVP-L were introduced. For more information, the interested reader may refer to [2] and [4].

4.4.5 Assessment with Respect to the Defined Criteria

Table 4.2 describes the four characteristics of MVP-L as well as the satisfaction of the eight requirements R1–R8, based on a subjective assessment. In this context, (+) represents full, (O) partial, and (–) no fulfillment of the respective requirement.

4.5 Software Process Engineering Metamodel

4.5.1 Overview

The first version of the SPEM standard was introduced by the Object Management Group (OMG) in 2002 and was built upon UML 1.4. It was revised in 2005 and again in 2007, when major changes led to version SPEM 2.0, which is compliant with UML 2. Due to UML compliance, standard UML diagrams such as activity diagrams or state chart diagrams can be used for visualizing processes models.

Table 4.2 MVP-L characteristics and requirements

Characterization: MVP-L	
Improvement	Proscriptive
Guidance	Multiperson
Requirements satisfaction: MVP-L	
R1—Natural representation	+
R2—Support of measurement	+
R3—Tailorability of models	O
R4—Formality	O
R5—Understandability	O
R6—Executability	+
R7—Flexibility	+
R8—Traceability	O

The development of SPEM was motivated by the abundance of different concepts for process modeling and software process improvement. These different concepts are usually described in different formats using different notations. Since achieving consistency between different approaches became increasingly difficult, the need for standardization arose. The SPEM standard for modeling software development processes has the following characteristics:

“The Software and Systems Process Engineering Meta-Model (SPEM) is a process engineering metamodel as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes.” [6]

4.5.2 Concepts

In the following sections, the basic SPEM concepts will be introduced. The conceptual framework of SPEM will be discussed, as will the basic notation constructs and the structure of the SPEM standard.

4.5.2.1 Conceptual SPEM Framework

The conceptual framework of SPEM mainly summarizes the aims of the standard. These are, on the one hand, to provide an approach for creating libraries of reusable method content and, on the other hand, to provide concepts for the development and management of processes. The combination of these two basic goals is seen as a solution that enables the configuration of more elaborate process frameworks and finally their enactment in real development projects (Fig. 4.9, based on [6]).

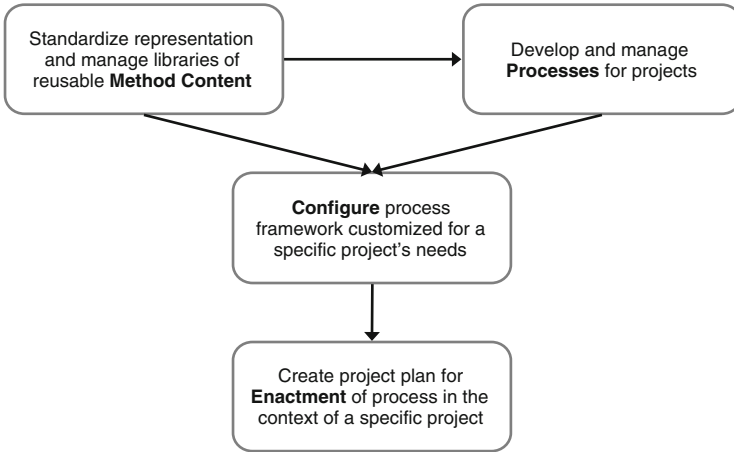


Fig. 4.9 SPEM 2.0 conceptual framework

As depicted in Fig. 4.9, the conceptual SPEM framework consists of four main elements: method content, processes, configuration, and enactment.

Libraries of method content address the need to create a knowledge base containing methods and key practices of software development. Method content captures key practices and methodologies in a standardized format and stores them in adequate libraries. This allows creating and managing reusable practices and methodologies. Such standardized content enables inclusion and integration of external and internal method content according to the current development requirements of an organization, and thus provides methodological support throughout different lifecycle development stages. Furthermore, the standardized method content elements can be used as a basis for the creation of custom processes.

The creation of processes can be supported based on the reusable method content. Processes can be defined as workflows and/or breakdown structures and, within this definition, the selected method content is supposed to be adapted to the specific project context. SPEM intends to provide support for the systematic development and management of development processes as well as for the adaptation of processes to specific project context.

As no two development projects are exactly alike, there is a need for tailoring specific processes from the organization's set of standard processes. With the element of configuration, SPEM aims at addressing concepts for the reuse of processes, for modeling variability, and for tailoring, thus allowing users to define their own extensions, omissions, and variability points on reused processes.

In order to support the enactment of processes within development projects, processes need to be instantiated in a format that is ready for enactment with a "process enactment system" (e.g., project and resource planning systems, workflow systems). Although SPEM 2.0 provides process definition structures, which allow

process engineers to express how a process shall be enacted within such an enactment system, support for enactment is generally regarded as weak [7].

4.5.3 Notation Constructs

The central idea of the SPEM is that a software development process is a collaboration between abstract active entities called *process roles*, which perform operations called *tasks* on concrete entities called *work products* [8].

The associations between role, task, and work product are shown in Fig. 4.10. Tasks are performed by one or more roles. Furthermore, tasks require one or more work products as input. They produce one or more work products as output. A role is responsible for one or more work products.

As described within the conceptual framework, SPEM uses an explicit distinction between method content and process, and the basic three entities must therefore be defined for both approaches.

The SPEM *method content* represents a library of descriptions of software engineering methods and best practices. It defines the “who, what, and how” of work increments that have to be done.

A SPEM *process* represents descriptions of coherent process steps, which enable performance of a certain task. It defines the “when” of work increments that have to be done.

Method content elements can be defined by using work product definitions, role definitions, and task definitions. Furthermore, Category and Guidance can be used. Guidance represents supporting resources, such as guidelines, whitepapers, checklists, examples, or roadmaps, and is defined at the intersection of method content and process because Guidance can provide support for method content as well as for specific processes. Table 4.3 gives an overview and description of basic notation construct elements belonging to Method Content.

Figure 4.11 shows an example representing a tester and all the tasks he performs (create test case, implement test, perform test) as well as the work products he is responsible for (test case, test log) within the software development process.

For the description of a Process, activities are mainly used as the major structuring element. Activities can be nested in order to define work breakdown structures or related to each other in order to define a flow of work. Furthermore, activities may have references to method content elements. These references refer to explicit method content by “using” the concepts *Task Use*, *Role Use*, and *Work Product Use*. Table 4.4 gives an overview and a description of basic notation construct elements belonging to Process.

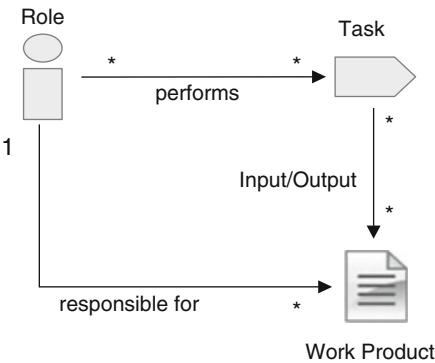


Fig. 4.10 Core method content concepts of role, task, and work product

Table 4.3 Key elements used for method content

Element	Description
Work product definition	Defines any artifact produced, consumed, or modified by a task. Work products can be composed of other work products. Examples: document, model, source code
Role definition	Defines a role and thus related skills, competencies, and responsibilities of one person or many persons. Is responsible for one or many work product(s) and performs one or many task(s). Examples: software architect, project manager, developer
Task definition	Defines work being performed by one or many role(s). A task has input and output work products. Inputs are differentiated into mandatory and optional inputs. Tasks can be divided into steps that describe subunits of work needed to perform the task
Category	Category is used for structuring other elements
Guidance	Can be associated with any SPEM model element to provide more detailed information about the element. Examples: checklist, template, example, guideline

4.5.4 Assessment with Respect to the Defined Criteria

Table 4.5 describes the four characteristics of SPEM 2.0 as well as the fulfillment of the eight requirements R1–R8, based on a subjective assessment. In this context, (+) represents full, (O) partial, and (–) no fulfillment of the respective requirement.

4.6 Tools for Software Process Modeling

Practitioners and process engineers are in need of software support for process modeling in order to be able to deal efficiently with process model creation and the administration of changes and modifications. For example, typical process guidelines

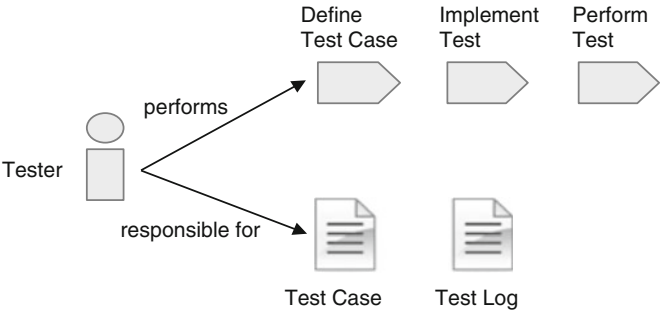


Fig. 4.11 Example for the role “Tester” with related elements

Table 4.4 Key elements used for process	
Element	Description
Work product use	Instance of a work product defined within Method Content. Can be used multiple times within process context
Role use	Instance of a role defined within Method Content. Can be used multiple times within process context
Task use	Instance of a task defined within Method Content. Can be used multiple times within process context. Additionally, definition of task-specific steps can be performed
Activity	Activities can be used to define work breakdown structures or workflows and thus group tasks within a software development process, which can then be used multiple times. Activities are used to model software development processes
Process Guidance	Can be used for structuring subprocesses by associating activities or tasks to it Can be associated with any SPEM model element to provide more detailed information about the element. Examples: checklist, template, example, guideline

are not only extensive but also cross-referenced, and, consequently, changes in certain areas lead to changes in other parts. Support is therefore useful for maintaining consistency. Such supporting software can have different functionalities. In order to be able to compare different solutions, the introduction of a reference model is useful. Therefore, in the first part of this section, the ECMA/NIST Reference Model for Software Engineering Environments will be introduced, which provides a framework for the discussion of different Software Engineering Environments (SEE). The second part will give an overview of the Eclipse Process Framework (EPF) and especially of the EPF Composer, as a specific tool for process modeling.

4.6.1 The ECMA/NIST Reference Model

The ECMA/NIST Reference Model for Frameworks of Software Engineering Environments was developed jointly by ECMA (European Computer Manufacturers

Table 4.5 SPEM 2.0 characteristics and requirements

Characterization: SPEM 2.0	
Improvement	Proscriptive
Guidance	Multiperson
Requirements Satisfaction: SPEM 2.0	
R1—Natural representation	+
R2—Support of measurement	O
R3—Tailorability of models	+
R4—Formality	O
R5—Understandability	+
R6—Executability	—
R7—Flexibility	O
R8—Traceability	O

Association) and NIST (National Institute of Standards and Technology, USA). The reference model provides a framework for describing and comparing different Software Engineering Environments (SEE) or Computer Aided Software Engineering (CASE) Tools [9]. As such, it is not a standard, but should help to identify emerging standards. In order to promote comparability, different services are grouped in this framework. These services are Object Management Services, Process Management Services, Communication Services, User Interface Services, Operating System Services, Policy Enforcement Services, and Framework Administration Services.

Furthermore, tools (respectively tool slots) are provided, which represent software that is not part of the SEE platform but uses services of the platform and can add further services to the platform. Based on [9], Fig. 4.12 displays an overview of the ECMA/NIST reference model.

In the following, the services that provide the core functionalities that a SEE should implement in some way are described in more detail (based on [9]):

- Object Management Services: The objective of these services is the definition, storage, maintenance, management, and access of object entities and of the relationships they have with other objects.
- Process Management Services: The objective of these services is the definition and computer-assisted performance of software development activities throughout the whole software lifecycle. As this service group addresses processes, the specific services will be described below in more detail. They are:
 - Process Development Service (process modeling)
 - Process Enactment Service
 - Process Visibility Service
 - Process Monitoring Service
 - Process Transaction Service
 - Process Resource Service
- Communication Services: The objective of these services is to provide information exchange among the services of an SEE.
- User Interface Services: These services are designed to allow interaction between the user and the SEE.

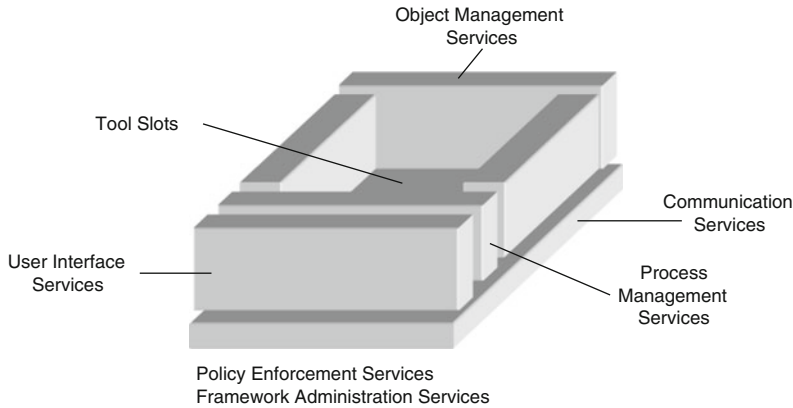


Fig. 4.12 ECMA/NIST reference model

- Operating System Services: These services provide descriptions for and integration with operation systems on which the SEE can be realized.
- Policy Enforcement Services: The purpose of these services is to provide security in an SEE.
- Framework Administration Services: These services provide support for constant adaptation of changes for the SEE.

All these service groups are further refined into specific services within the reference model, but a detailed discussion of all services is beyond the scope of this section. After this brief overview of the reference model, a closer examination of the six services from the Process Management Services group will be provided below (based on [9]):

The *Process Development Service* as described by ECMA/NIST shall enable the modeling of processes within the SEE. Therefore, a form for documenting the process models should be defined, and operations for the creation, modification, and deletion of process models should be included. The formalism of the process description is not restricted, thus allowing informal process description in natural language as well as the use of formal process modeling notations.

The *Process Enactment Service* should facilitate control and support for the enactment of processes defined in the SEE. The operations that are regarded as appropriate in this context are:

- Instantiation of process definitions
- Linking together of process elements
- Enactment of instantiated process definitions
- Suspension and restart of an enacting process
- Abortion of an enacting process
- Tracing of an enacting process
- Checkpoint and rollback
- (Dynamic) Change of enacting instances of a process definition

The *Process Visibility Service* aims at the definition and maintenance of visibility information, by defining which information should be visible to other entities and when and where it should be visible. Operations that are regarded as appropriate for visibility are:

- Establishing access to specified information for an entity
- Hiding information from other entities
- Defining and managing visible areas and communication structures
- Visualizing defined areas and communication structures
- Controlling access rights

The *Process Monitoring Service* observes the evolving process states, detects the occurrence of specified process events, and enacts necessary actions based on observation and detection. In this context, the definition of specific process events and derived actions should be supported. Relevant operations are:

- Definition, modification, deletion, and querying of event definitions
- Manipulation of the control process definitions
- Attaching/detaching actions to/from events
- Querying of the state of a process

The *Process Transaction Service* provides support for the definition and enactment of process transactions, which can be understood as process elements composed of a sequence of atomic process steps. Such a transaction should be either completed entirely or rolled back to the preenactment state. Appropriate operations, which are described in this context, consist of:

- Creation, initiation, abortion, deletion, modification of transactions
- Commit of transactions
- Checkpoints and rollback of process states
- “Login” and “logout” of long-lived transactions

The *Process Resource Service* accounts for allocation and management of resources during enactment of a defined process. The operations defined for the Process Resource Service are:

- Definition, creation, modification, and deletion of process resource types and resource entities
- Mapping of project resources to resource model instances
- Mapping of resource model instances to process instances
- Adaptation of mapping

In this section, an overview of the ECMA/NIST Reference Model for Software Engineering Environments was given, which is useful for describing and comparing tools for process management and its functionalities. In the following section, one example of a tool that supports process modeling will be given by introducing the Eclipse Process Framework and the EPF Composer as well as its main functionalities.

4.6.2 *The Eclipse Process Framework (EPF) Composer*

The Eclipse Process Framework (EPF) is an open-source project within the Eclipse Foundation and was initiated in January 2006. The EPF project has two main objectives. The first objective is to provide an extensible framework and exemplary tools for software process engineering. This includes support for method and process authoring, library management, configuration, and publishing of processes. The second objective is to provide exemplary and extensible process content for a range of software development and management processes, and thereby support a broad variety of project types and development styles [10].

The EPF Composer has been developed in order to fulfill the first objective. Its conceptual framework is based on SPEM 2.0, and for this reason, the aforementioned concepts in the section about SPEM are useful for understanding the functionality of this tool. The EPF Composer is equipped with predefined process content, which addresses the second objective. The process framework provided with the EPF Composer is called Open Unified Process, and is strongly influenced by IBM's Rational Unified Process [10]. As it was not the aim of the project to provide a process framework, this process content can be understood as a suggestion. In the meantime, further process content has been provided (e.g., agile practices).

4.6.2.1 Basic Concepts

According to SPEM 2.0, the EPF Composer¹ [10] implements the distinction between Method Content and Process, providing capabilities for creating method libraries.

Using the authoring capabilities of EPF Composer, method content can be defined. This definition of method content resembles the creation of method content libraries according to SPEM 2.0.

Tasks are a main element of method content. For tasks, a description can be provided that contains general information. It is also possible to provide detailed information and versioning. Tasks can be refined into the steps that should be performed during task performance. These steps can be defined, ordered, and their content can be described in detail. Moreover, the associated roles, work products, and guidance can be added. Those are defined and described as separate entities within the method content library and then related to a task during task definition. This approach represents an implementation of the task concept provided by SPEM 2.0.

The EPF Composer addresses mainly two audiences. By providing authoring capabilities, it addresses process authors/engineers and provides them with a tool for creating and publishing method content and processes. Simultaneously, it provides functionalities that address process consumers/users by integrating the

¹ The content presented here is based on the EPF Composer Version 1.5.0.3.

possibility to publish content in the form of websites that can be browsed. There, the user can find necessary information concerning processes, methods, and guidance (checklists, concepts, guidelines, etc.).

In addition to Method Content, EPF Composer provides capabilities for process definition and adaption. Similar to method content, processes can be authored and published. Within the authoring view, new processes can be composed by creating a sequence of tasks that were already defined in method content. In this way, tasks are integrated that contain associated roles and work products. During process composition, the predefined tasks can be modified, and it is therefore possible to tailor specific processes from predefined method content. Furthermore, it is also possible to tailor previously defined process compositions. The concepts of method content and process variability will be discussed in more detail in the next section.

4.6.2.2 Method Variability

Method variability provides the capability of tailoring existing method content without directly modifying the original content. This is an important ability, as future updates might lead to inconsistencies in the dataset. Variability can be used, for example, to change the description of a role, to add/change steps to an existing task, or to add/change guidance.

The concept used is similar to inheritance in object-oriented programming. Thus, it allows reuse of content with further specialization/modification. For realizing this concept, the EPF Composer uses “plug-ins.” After such a plug-in has been created, it can be defined which existing content should be “inherited.”

There are four types of method variability [10]:

- **Contribute:** The contributing element adds content to the base element. The resulting published element contains the content of the base element and the contributing element.
- **Replace:** The replacing element replaces the base element. The resulting published element is the replacing element.
- **Extend:** The extending element inherits the content of the base element, which can then be specialized. Both the base element and the extending element are published.
- **Extend and Replace:** Similar to extend, but the base element is not published.

4.6.2.3 Process Variability

Concepts for process variability are based on activities, which are the elements used to compose processes. Activity variability is based on the same four types of variability as method content (see above: contribute, replace, extend, and extend and replace).

Additionally, activities may be used to create capability patterns. Capability patterns can be defined as a special type of process that describes a reusable cluster of activities for a certain area of application/interest. Processes can be created by using capability patterns in the following ways [10]:

- Extend: The process inherits the properties of the capability pattern. Updates to the capability pattern or respective activities are also realized in the respective process.
- Copy: A process is created based on a copy of the capability pattern. In contrast to extend, the respective process is not synchronized with the capability pattern when changes occur.
- Deep Copy: Similar to copy, but is applied recursively to activities of the respective capability pattern.

References

1. Rombach HD, Verlage M (1995) Directions in software process research. In: Zelkowitz MV (ed) *Advances in computers*, vol 41. Academic Press, Boston, MA
2. Bröckers A, Lott CM, Rombach HD, Verlage M (1995) MVP-L language report version 2, Technical Report Nr. 265/95, University of Kaiserslautern, Department of Computer Science, Software Engineering Chair
3. Parnas D (1972) On the criteria to be used in decomposing systems into modules. *Commun ACM* 15(12):1053–1058
4. Bröckers A, Differding C, Hoisl B, Kollnischko F, Lott CM, Münch J, Verlage M, Vorwieger S (1995) A Graphical Representation Schema for the Software Process Modeling Language MVP-L, University of Kaiserslautern, Department of Computer Science, Software Engineering Chair
5. Rombach HD (1991) MVP-L: a language for process modeling in-the-large. University of Maryland, College Park, MD
6. Object Management Group (2008) *Software & systems process engineering meta-model specification version 2.0*. OMG, Needham, USA
7. Bendraou R, Combemale B, Crogut X, Gervais M (2001) Definition of an Executable SPEM 2.0. In: *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, Nagoya, Japan, 5–7 Dec 2007. doi: 10.1109/ASPEC.2007.60
8. Object Management Group (2005) *Software process engineering meta-model specification version 1.1*. OMG, Needham, USA
9. ECMA/NIST (1993) *Reference model for frameworks of software engineering environments*, Technical Report ECMA TR/55
10. Eclipse Foundation (2009) EPF composer: open UP library. http://www.eclipse.org/epf/downloads/tool/tool_downloads.php. Accessed 27 Jun 2011

Software Process Definition and Management
Münch, J.; Armbrust, O.; Kowalczyk, M.; Soto, M.
2012, XX, 236 p., Hardcover
ISBN: 978-3-642-24290-8