Turning Method Engineering Support into Reality

Mario Cervera, Manoli Albert, Victoria Torres, and Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software,
Universidad Politécnica de Valencia, 46022 Valencia, Spain
{mcervera,malbert,vtorres,pele}@pros.upv.es

Abstract. The Situational Method Engineering (SME) discipline emerged two decades ago to face up to the challenge of the in-house definition of software production methods and the construction of the corresponding supporting tools. However, nowadays most of the existent proposals only focus on one of the phases of the SME lifecycle. In order to fill this gap, in this paper we present a methodological framework that equally encompasses two of these phases, which refer to the method design and implementation. In order to support them in an effective manner, we advocate for the use of the Model Driven Development (MDD) paradigm. Applying these ideas, the framework has been defined on top of a MDD infrastructure based on meta-modeling and model transformation techniques. In addition, we provide implementation details of the framework in an Eclipse-based modeling platform, namely MOSKitt.

Keywords: Method Engineering, Model Driven Development, CAME Environment, Eclipse, MOSKitt.

1 Introduction

Software Production Methods (hereafter simply methods) are organized and systematic approaches for software development, which can adequately govern the disciplined execution of real software development projects, and are composed, inter alia, of structured and integrated sets of activities, work products and roles. Since the definition of a universally applicable method has for long been considered unattainable, it is necessary to find solutions that enable the in-house specification of methods adapted to specific context needs and the construction of the corresponding supporting tools. Up to now, the SME discipline seems to be the most promising alternative to supply this need.

The SME discipline constitutes a sub-area of a broader field called Method Engineering (ME). Specifically, within the ME (and SME) field, method and software engineers mainly deal with (1) the definition of methods (method design) and (2) the construction of the supporting software tools (method implementation)[1]. Therefore, proposals aimed at supporting ME should cover these two phases of the ME process. However, most of the ME proposals existing in the literature (and their corresponding

[1] Other tasks such as the analysis of the method requirements and the validation of the method are also part of the Method Engineering discipline but are outside of the scope of this paper. These tasks will be considered in future work.

tools) only focus on one of them. As examples of this reality we find Computer Aided Method Engineering (CAME) and metaCASE environments. On the one hand, CAME environments generally focus on the method design phase, supporting the specification of project-specific methods for software development. In some cases, these specifications are used for building CASE tools, but with very limited capabilities. On the other hand, the so-called metaCASE environments generally focus on the method implementation, supporting the customization of CASE tools by means of high level specifications. These specifications normally define the modeling languages that are to be supported by the CASE tool and, sometimes, also the process that establishes the order in which these languages must be used. Thus, these specifications are oriented towards CASE tool definition and therefore they do not represent complete software production methods.

In order to provide a more complete proposal, in this paper we propose a methodological framework that equally encompasses the method design and method implementation phases. Combining these two phases brings an important benefit. It increments the method specifications' value in terms of how much functionality is derived from them. That is to say, these specifications are not only used for governing the execution of the software development projects, but also for the construction of CASE tools that support the methods and assist the software engineers in the development of the final systems. To achieve this goal in an effective manner, we find crucial to define an infrastructure that (1) allows the method engineer to define methods that can be applied in real software projects and also (2) (semi)automates the construction of tools that provide adequate support to the specified methods. To successfully face the definition of this infrastructure, we advocate for the use of the MDD paradigm. Thereby, we have defined a MDD infrastructure based on meta-modeling and model transformation techniques that lays the foundations of the methodological framework. Specifically, the meta-modeling techniques are based on the Software & Systems Process Engineering Meta-model (SPEM) [30] and are the means that allow the method engineer to carry out the method design. On the other hand, model transformations (semi)automate the performance of the method implementation. By applying these ideas, we have defined a methodological approach that not only tackles the definition of methods following a widely accepted standard (SPEM), but also proposes to use these definitions for the (semi)automatic generation of tools that provide rich support to the methods (textual and graphical editors, code generators, model transformations, process enactment support, etc.).

The work reported here is an extension of our previous works [7] and [8]. On the one hand, the theoretical part of the methodological framework is analyzed in depth, with a contextualization of the different parts of the framework. On the other hand, the software infrastructure of the framework has evolved by enhancing the way in which engineering tools assist method engineers during the method construction.

Furthermore, as a proof of concept, we also provide details of the implemented framework, which has been developed on top of MOSKitt [21], an Eclipse-based modeling platform whose plugin-based architecture and integrated modeling tools turn it into a suitable platform to support the proposal.

The remainder of the paper is structured as follows. First, section 2 summarizes the state of the art. Then, section 3 provides an overview of the proposal. Section 4 and 5 thoroughly detail the MDD infrastructure and the methodological framework respectively. Finally, section 6 draws some conclusions and outlines future work.

2 State of the Art

The term Method Engineering was first introduced in the mid-eighties by Bergstra et al in [4]. Since then, many works developed both at academia and industry have contributed to this field. In order to underpin its theory, a survey of the last strands in ME is gathered in [17]. In this work, the definition proposed by Brinkkemper et al. in [5] is used to define ME as *the engineering discipline to design, construct and adapt methods, techniques and tools for the development of information systems (IS).*

Considering this definition, we have found that there are proposals in the ME literature that mainly focus on (1) the design, construction and adaptation of methods (i.e. the method design) while others concentrate on (2) the techniques and tools for supporting such methods (i.e. the method implementation). On the one hand, among the proposals mostly dedicated to method design, we find proposals such as Brinkkemper's [5, 6], Ralyté's [20, 24] or Henderson-Sellers' [15], which tackle the method construction by means of the assembly of method fragments or chunks stored in a method base repository. Examples of tools that fall in this first category are MERET [18], Method Editor [29] and Decamerone [14]. Some of these proposals do support the generation of CASE environments but with limited capabilities. For instance, Method Editor enables the generation of tools that include a series of diagram editors that allow the software engineer to create/manipulate the products specified in the method. However, Method Editor does not support the specification of automated tasks that require the inclusion of a model transformation in the generated tool. Thus, these CASE tools lack code generation capabilities.

On the other hand, there are proposals that mostly focus on the method implementation [10, 12, 28]. These are the so-called metaCASE environments that generally support the construction of CASE tools. Examples of tools that fall in this category are MetaMOOSE [10], KOGGE [28] and MetaEdit+ [19]. For instance, MetaEdit+ [19] provides a specification language (called GOPPRR) that is oriented towards the definition of the abstract syntax of the modeling languages (in [19] called "methods") that need to be supported by the resulting CASE tool. In contrast, in our proposal we provide a full methodology that assist in the definition of complete software production methods by means of the SPEM standard, and also proposes the use of a meta-meta-model (such as GOPPRR) for the definition of the modeling languages that enable the creation of the method products (see sections 3 and 5). In particular, the meta-meta-model that is used in the CAME environment that supports our proposal is Ecore.

After studying all the aforementioned proposals, we have found an important lack of software tools that provide complete support to ME. In this paper, we advocate for the use of the MDD paradigm as a way to improve this situation. In particular, we define a methodological framework that is being implemented in the context of the MOSKitt platform [21] and, by applying MDD techniques, equally supports the method design and the method implementation phases.

3 Overview of the Proposal

In order to provide an overview of the proposal, in this section the methodological framework is briefly introduced. The three phases that compose the framework are: method design, method configuration and method implementation (see Fig. 1).

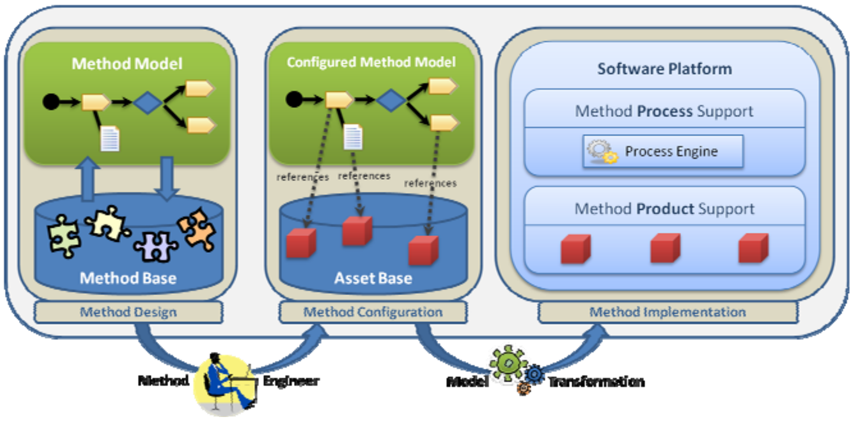


Fig. 1. Methodological framework overview

- *Method design*: during this phase, the method engineer builds the method specification as a model (hereafter the method model) using the SPEM standard [30]. This model can be built from scratch or reusing method fragments stored in a *Method Base* repository that has been implemented following the RAS standard [26]. The built model constitutes a first version of the method that does not include details about the technologies and notations that will be used during the method execution. For instance, the method engineer can specify a generic product called “Business Process Model”, without stating in which notation this product will be created when the method is executed.
- *Method configuration*: in this phase, the method model is instantiated with the specific technologies and notations that will be used during the method enactment. This instantiation is achieved by associating tasks and products with editors, metamodels, transformations, etc. that are stored in a repository called *Asset Base* (implemented following the RAS standard). For instance, the product “Business Process Model” can be associated with a “BPMN editor”. Thus, the method engineer is indicating that this editor must be included in the generated tool, so that it enables the manipulation of this particular product. The main benefit of separating method design and configuration is that we keep generic definitions of methods (which means that we can take this generic definition and perform different method configurations), stressing the importance of reusability.
- *Method implementation*: in this phase, the method model is used as input of a model transformation that generates the tool support. This tool provides support to the product and process parts of the method[2]. The product support consists of the tools that enable the creation/manipulation of the method products (i.e. the resources associated to the method elements in the previous phase). The process support consists of a process engine that enables the method process execution.

[2] The product part represents the artifacts that must be built during the method execution and the process part consists of the procedures that must be followed to build such products.

4 The MDD Infrastructure

In this section we present the MDD infrastructure that lays the foundations of the methodological framework. As mentioned above, this infrastructure is based on meta-modeling and model transformation techniques.

4.1 Meta-modeling

Meta-modeling has always played a key role in the ME field as it allows the definition at a high level of abstraction of the concepts, constraints and rules that are applicable in the construction of methods. In general, proposals focusing on the method design use meta-modeling as their underlying technique to define methods [6, 18, 20]. Moreover, proposals focusing on the method implementation use these techniques to specify the modeling languages supported by the generated tools [12, 19, 28].

In our proposal we use meta-modeling techniques for the creation of the method model, in particular following the SPEM standard. A study about the applicability of SPEM to ME is presented in [22]. In this work, the authors present some of the SPEM advantages and disadvantages for supporting the method design. Among the SPEM advantages we highlight: (1) wide acceptance in the field of process engineering, (2) good ME process coverage, (3) support to both product and process parts of methods and (4) good abstraction and modularization. Regarding its disadvantages, [22] points out the lack of executable semantics, but proposes to overcome this limitation by using a model transformation to transform the process models into executable representations that can be executed by workflow engines.

In order to provide a more in-depth view on how the SPEM meta-model is used in our proposal, below the structure of the method fragments from which SPEM models can be assembled is presented in detail. In general, in the ME proposals that suggest the use of method fragments, these are obtained by instantiating some class of a meta-model. For instance, in the OPEN Process Framework [11] method fragments are generated by instantiation from one of the top levels classes: Producer, Work Product and Work Unit [17]. Specifically, next subsection details the SPEM classes from which method fragments can be created and, furthermore, it presents a taxonomy that classifies the different types of fragments that are used in the proposal.

Method Fragments. We use the term *method fragment* to denote the atomic element from which methods can be assembled. Other terms to name these atomic elements, such as method chunk, have been proposed in the ME literature [16]. A method fragment can be either a *product fragment* (instances of meta-classes that represent products) or a *process fragment* (instances of meta-classes that represent processes). This differentiation allows us (1) to leverage the separation between product and process specification provided by SPEM[3], (2) to relate one process fragment with many

[3] In order to use the same terminology as the used in the ME field, in our proposal we consider analogous the product-process separation of methods and the SPEM separation between method content and method process.

product fragments, and (3) to reuse one product fragment in the definition of many process fragments.

Attending to the different phases identified in our framework (see section 3), we use a third type of fragment, namely *technical fragment*, term that was first proposed in [13]. In our proposal, these fragments contain the tools that are associated to the products and tasks of the method during the method configuration and that make up the infrastructure of the generated CASE tools.

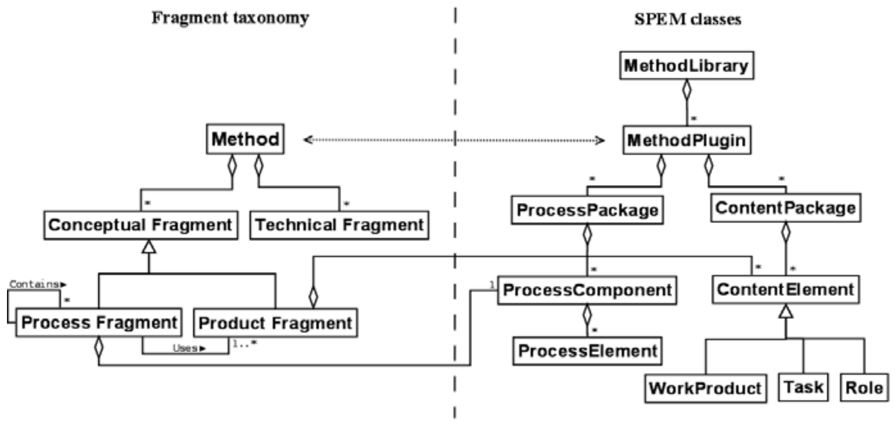


Fig. 2. Relationship between method fragments and SPEM classes

In order to illustrate the hierarchical organization of the various types of fragments, the left side of Fig. 2 graphically presents our fragment taxonomy. In this taxonomy, the new abstract category *conceptual fragment* (also proposed in [13]) is introduced for grouping product and process fragments. Moreover, additional information has been included, e.g. the relationship *Contains* which represents that SPEM processes can contain nested subprocesses, or the relationship labeled as *Uses* which represents that one process fragment can reference from one to many product fragments.

On the other hand, the right side of Fig. 2 shows a simplified view of the SPEM meta-model. In SPEM, a method is represented by a *MethodPlugin*. Each *Method-Plugin* contains both *ContentPackages* and *ProcessPackages*. *Tasks*, *Roles* and *WorkProducts* are stored in *ContentPackages*. Similarly, within *ProcessPackages*, processes are stored as instances of the class *ProcessComponent*.

Note that some of these SPEM concepts have been associated with fragments of our taxonomy. These associations illustrate a containment relationship. For instance, process fragments are associated with one *ProcessComponent*. Thus we are representing that, when process fragments are stored in the repository, they contain a SPEM model that includes one instance of the class *ProcessComponent*. Furthermore, product fragments are associated with *ContentElements*, which represents that these fragments can contain any instances of *Task*, *Role*, and *WorkProduct*.

Finally, even though it has been omitted in Fig. 2, method fragments are defined by a series of properties that enable their later retrieval from the repository. The fragment properties are stored in the manifest file of the RAS asset that embodies the fragment. Specifically, we make use of some of the properties defined in [23]. According to these properties, our method fragments are characterized by:

- *Descriptor*: it contains general knowledge about the fragment. For now, we consider the attributes *origin*, *objective* and *type*. Some examples of valid types in our proposal are *task*, *role* and *work product* for product fragments that contain atomic elements, or *meta-model*, *editor*, *model transformation* and *guide* for technical fragments (see section 5.2).
- *Interface*: it describes the context in which the fragment can be reused. For now, we only consider the attribute *situation.*

4.2 Model Transformations

In the previous subsection we showed that the application of meta-modeling in the ME field is not new. However, we find that the ME approaches that make use of these techniques do not really take full advantage of the possibilities that MDD offers. As stated in [3], "*MDD improves developers' short-term productivity by increasing the value of primary software artifacts (i.e. the models) in terms of how much functionality they deliver*". Following this statement and contrary to what current ME approaches do, we want to leverage models going one step further. Defining the method as a model and considering this model as a software artifact allows us to face the implementation of the CASE tool generation process by means of model transformations.

In particular, these transformations have been implemented in the CAME environment that supports our proposal as a single model-to-text (M2T) transformation using the XPand language [31], which is the language used within the context of the MOSKitt project [21] for that purpose. Further details about this M2T transformation are provided in section 5.3.1 and in [8].

5 The Methodological Framework

In this section, we detail the phases in which the methodological framework has been designed. For each of these phases, we provide first a generic description and then we detail the software infrastructure that has been implemented in MOSKitt to support it.

5.1 Method Design

During the method design the method model is built using SPEM. The construction of this model is performed by means of a combination of two approaches proposed in [24]: (1) the paradigm-based and (2) the assembly-based. In order to illustrate how these approaches are applied in our framework, we use the *Map* process meta-model proposed in [27]. Following this meta-model, processes are represented as labelled directed graphs with intentions as nodes and strategies as edges between intentions.

The Paradigm-Based Approach. In Fig. 3 we show how the method model is built in our proposal following the paradigm-based approach. The hypothesis of this approach is that the new method is obtained either by abstracting from an existing model or by instantiating a meta-model. This starting model is called the *paradigm model*. Specifically, we build the method models by instantiating a meta-model (i.e. the SPEM meta-model).

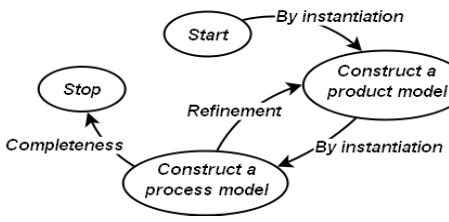


Fig. 3. Paradigm-based approach (adapted from [24])

As shown in the figure, the construction of the method model is performed in two steps: first, the method engineer builds the *product model* (i.e. the products, roles, etc. that compose the SPEM method content). Secondly, the method engineer builds the *process model* (i.e. the process component that composes the SPEM method process). In addition, backtracking to the construction of the product model is possible when building the process model thanks to the refinement strategy.

The Assembly-Based Approach. Fig. 4 shows how the assembly-based approach is carried out in our proposal. This process is followed when the method engineer wants to reuse product or process fragments stored in the Method Base.

As shown in the figure, the method engineer starts by specifying the requirements of the fragments to be retrieved. These requirements are specified as queries that must be formulated by giving values to the method fragment properties (see section 4.1).

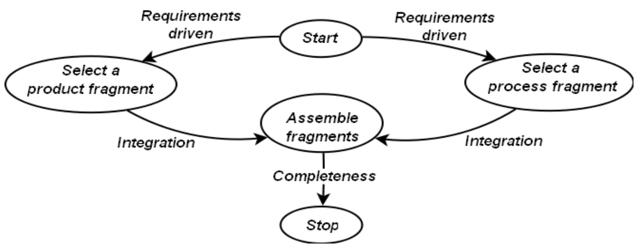


Fig. 4. Assembly-based approach (adapted from [24])

As an example, a query for retrieving a product fragment containing a *task* for *system specification* may include parameters as follows:

```
Type = 'Task' AND Objective = 'System Specification'
```

Once the fragments have been obtained[4], the intention "Assemble fragments" must be achieved by means of the "integration" strategy. This strategy consists of the integration of the selected fragments into the method model (considered here as a process fragment of a higher level of granularity). Depending on the type of the fragment this integration varies. For product fragments, the tasks, roles etc. are directly included in a *ContentPackage*. For process fragments, the process elements are included as a subprocess in the method under construction.

Finally, note that during the method design new fragments can be created for their later reuse during the construction of other methods. In order to illustrate how product and process fragments are created, Fig. 5 shows the process that must be followed.

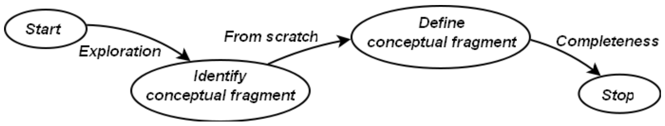


Fig. 5. Conceptual fragment creation (adapted from [25])

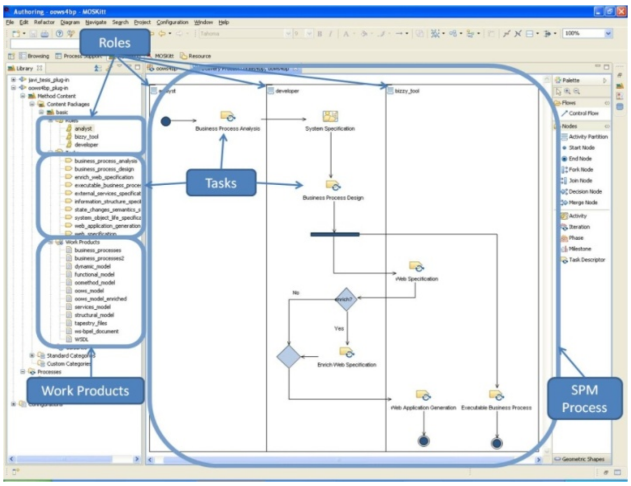
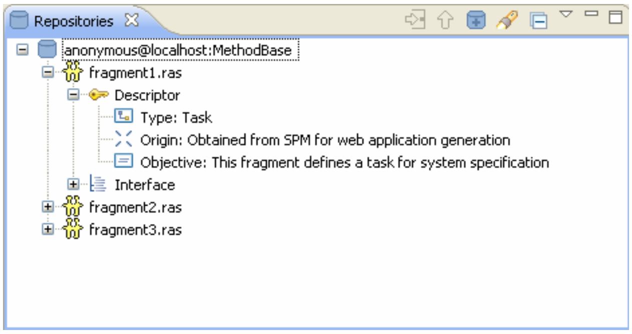


Fig. 6. EPF Composer editor in MOSKitt

[4] Note that if a process fragment is retrieved, then the associated product fragments are automatically selected. This is due to the one-to-many cardinality of the relationship between product and process fragments in figure 2.



anonymous@localhost:MethodBase
fragment1.ras
Descriptor
Type: Task
Origin: Obtained from SPM for web application generation
Objective: This fragment defines a task for system specification
Interface
fragment2.ras
fragment3.ras

Fig. 7. Repository client connected to the Method Base

First, the method engineer explores the method model to identify the elements that must be included in the conceptual fragment. These elements will be tasks, roles, etc. (for a product fragment) or a process component (for a process fragment). Then, the method engineer defines the fragment by giving values to the fragment properties. Once this process is completed, a RAS asset is created and stored in the Method Base.

Method Design Software Infrastructure. In order to provide software support within MOSKitt to the method design phase, the following tools have been integrated as Eclipse plugins:

- *A method editor*: in order to enhance MOSKitt with the capability of building method models, the EPF Composer (a SPEM 2.0 editor provided in the EPF Project [9]) has been integrated. This editor enables the enactment of the process described in Fig. 3, i.e. it allows method engineers to build SPEM models. In addition, it has been extended so it enables the enactment of the process shown in Fig. 5, i.e. it supports the creation of fragments. In Fig. 6[5], a screenshot of the EPF Composer integrated in MOSKitt is shown.
- *A repository client*: In order to reuse the fragments stored in the Method Base during the construction of the method model, it is necessary to implement a repository client that enables the enactment of the process described in Fig. 4. To do so, the repository client must allow the method engineer to (1) connect to the repository, (2) search and select conceptual fragments and (3) integrate them in the method model under construction. Fig. 7 shows the repository client that has been implemented in MOSKitt as an Eclipse view.
- *A guide to build the method model*: A guide is provided as an Eclipse cheatsheet to assist the method engineer in the performance of the method design phase.

5.2 Method Configuration

In this phase the method model is completed by including details about the technologies and notations that will be used during the method execution. Fig. 8 shows how this phase is performed. In particular, the method engineer specifies the requirements that are used to retrieve a technical fragment from the Asset Base. Once this is done, he/she associates it with a task or product of the method model.

[5] Also available at http://users.dsic.upv.es/~vtorres/moskitt4me/

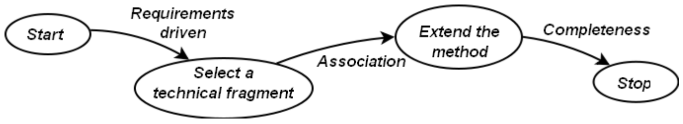


Fig. 8. Process model for asset association

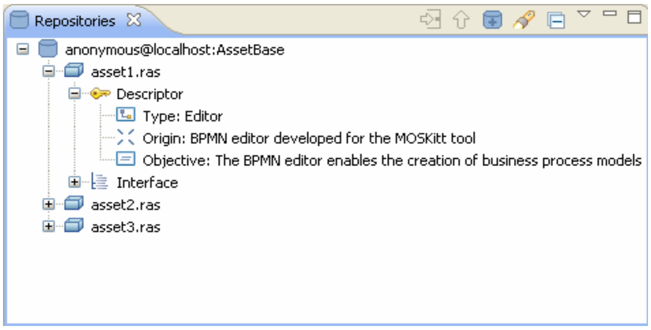
Note that it is possible that no suitable technical fragment is available in the repository. In case the method engineer considers that a new technical fragment must be created, a process similar to the one defined in Fig. 5. is followed. First, the required tool is implemented ad-hoc for the method under construction. For instance, in the CAME environment that supports our proposal these tools are implemented as Eclipse plugins developed using the CAME environment itself. Once the tool is implemented, the method engineer defines the technical fragment by giving values to the fragment properties. Then, a RAS asset is created and stored in the Asset Base.

We detail below the various types of technical fragments that can be stored in the Asset Base, to which elements they can be associated and for which purpose:

- *Meta-model*: meta-models can be associated to method products to specify the notation that will be used in the generated tools for their manipulation (e.g. the "BPMN meta-model" can be linked to the product "Business Process Model").
- *Editor*: textual/graphical editors can be associated to method products to specify the resource that will be used in the generated tools for their manipulation (e.g. a "BPMN editor" can be linked to the product "Business Process Model").
- *Transformation*: model transformations can be associated to tasks of the method. Thus, these tasks will be automatically executed in the final tool by means of the model transformations (e.g. a M2T transformation can be linked to the task "Generate report").
- *Guide*: guides (i.e. text files, process models, etc.) can be optionally associated to manual tasks of the method. These files will be included in the final tool and will assist software engineers in the performance of the tasks. For instance, a map can be associated to the task "Build Business Process Model" to define as a process model the steps that must be followed to perform the task.

Method Configuration Software Infrastructure. In order to provide software support within MOSKitt to the method configuration phase, the following tools have been integrated as Eclipse plugins:

- *A repository client*: In order to associate technical fragments with elements of the method model, it is necessary to implement a repository client that enables the enactment of the process described in Fig. 8. To do so, the repository client must allow the method engineer to (1) connect to the repository, (2) search and select technical fragments and (3) associate them with the elements of the method. The repository client of Fig. 7 can be reused for this purpose. Fig. 9 shows this repository client connected to the Asset Base.
- *A guide to configure the method model*: A guide is provided as an Eclipse cheatsheet to assist in the performance of the method configuration phase.



Repositories

anonymous@localhost:AssetBase
asset1.ras
Descriptor
Type: Editor
Origin: BPMN editor developed for the MOSKitt tool
Objective: The BPMN editor enables the creation of business process models
Interface
asset2.ras
asset3.ras

Fig. 9. Repository client connected to the Asset Base

5.3 Method Implementation

During this phase a tool supporting the method is obtained by means of model transformations. This tool is mainly divided into two parts: the dynamic part and the static part (see Fig. 10).

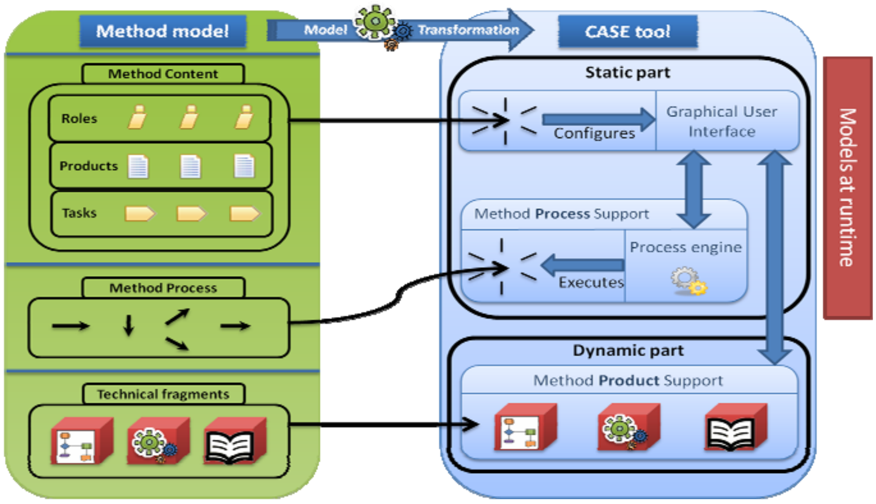


Fig. 10. Transformation mappings

The Dynamic Part. The dynamic part is composed of those elements that are directly obtained from the method model and are, thus, dependent on the specified method. In particular, these elements correspond to the tools that are in charge of providing software support to the product part of the method and make up the infrastructure of the tool (e.g. editors, model transformations, etc.). These tools are specified within the method model as *technical fragments*, which are stored as RAS assets that contain the implementations of the tools (e.g. the Eclipse plugins that implement a graphical editor). Therefore, the model transformation integrates these tools in the generated CASE environment.

The Static Part. The static part is composed of those elements that are always included in the final tool and, thus, their implementation is independent of the method. Even though the implementation of these components does not depend on the method model, they need to use this model at runtime[6]. Specifically, two components make up this part:

- The process engine: this component provides support to the process part of the method. It is always included in the generated tools and is in charge of the execution of the *method process* part of the SPEM model[7]. This execution conducts the orchestration of the different tools that allow the creation/manipulation of the method products (i.e. the technical fragments).
- The graphical user interface (GUI): the GUI is composed of those elements that make up the visual representation of the tool and allow software engineers to execute method instances by means of the process engine. The GUI of the generated CASE tools does not directly depend on the method model (so, they always have the same look & feel) but it uses the *method content* part of the SPEM model to configure itself. For instance, depending on the role selected by the user, the GUI filters its content to show only the products and tasks that the user is in charge of.

Method Implementation Software Infrastructure. In order to provide software support within MOSKitt to the method implementation phase, the following tool has been implemented and integrated as an Eclipse plugin:

- *A M2T transformation:* this transformation obtains the tool that supports the method specified in the method model. This tool corresponds to a MOSKitt reconfiguration that only contains the required Eclipse plugins to support the method (i.e. the plugins contained in the technical fragments[8], the process engine and the Eclipse views that compose the GUI). In order to build this MOSKitt reconfiguration we make use of the Eclipse Product Configuration files (`.product` files). This type of files gathers all the required information to automatically generate an Eclipse-based tool such as MOSKitt. So, considering that this tool is obtained from a `.product` file, the model transformation has been implemented as a M2T transformation. This transformation takes as input the model resulting from the method configuration phase and generates a `.product` file through which the final tool is automatically generated.

6 Conclusions and Future Work

In the ME field it is still unclear how to combine different subareas into a whole in order to define more complete proposals. As examples of this reality we find CAME and metaCASE environments, which either focus on the method design or the method

[6] Runtime in this context refers to the method execution in the generated CASE tool.

[7] SPEM does not have executable semantics. Therefore, a mapping between SPEM and an executable language is needed here. We are planning to tackle this issue in the future.

[8] The dependencies of these plugins must also be included. We are planning to tackle dependencies management in the future.

implementation phases of the ME process. In this work, we have detailed the different steps of a methodological framework that adequately covers these two phases. For this purpose, the proposed framework applies an MDD approach, tackling the method design by means of meta-modeling techniques based on the SPEM standard and the method implementation by means of model transformations.

The presented framework is being defined and implemented within the context of the MOSKitt project. This project constitutes a jointly work developed by the *Conselleria de Infraestructuras y Transporte* and the *Centro de Investigación en Métodos de Producción de Software* to develop a CASE tool to support the gvMétrica method. There is a big community involved in the project, ranging from analysts to end users, which are in charge of validating each new release of the tool. This setting constitutes an adequate environment to validate our proposal. In fact, in the near future we are planning to integrate our prototype into a MOSKitt version in order to use it for the definition of gvMétrica and the construction of the supporting tool.

Regarding future work, we are working on the improvement of the CAME environment that supports our proposal. For instance, we are planning the integration of a process engine such as Activiti [1]. Furthermore, we are concerning with one of the big challenges of ME [2], which deals with the variability of methods at modeling level and runtime. Providing support to variability will allow stakeholders to dynamically adapt methods and their supporting tools to changes that occur during method execution.

References

1. Activiti, `http://www.activiti.org/`
2. Armbrust, O., Katahira, M., Miyamoto, Y., Münch, J., Nakao, H., Ocampo, A.: Scoping Software Process Models - Initial Concepts and Experience from Defining Space Standards. In: ICSP, pp. 160–172 (2008)
3. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20, 36–41 (2003)
4. Bergstra, J., Jonkers, H., Obbink, J.: A Software Development Model for Method Engineering. In: Roukens, J., Renuart, J. (eds.) Esprit 1984: Status Report of Ongoing Work. Elsevier Science Publishers, Amsterdam (1985)
5. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information and Software Technology 38, 275–280 (1996)
6. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-Modelling Based Assembly Techniques for Situational Method Engineering. Inf. Syst. 24, 209–228 (1999)
7. Cervera, M., Albert, M., Torres, V., Pelechano, V.: A Methodological Framework and Software Infrastructure for the Construction of Software Production Methods. In: International Conference on Software Processes (2010)
8. Cervera, M., Albert, M., Torres, V., Pelechano, V., Cano, J., Bonet, B.: A Technological Framework to support Model Driven Method Engineering. Taller sobre Desarrollo de Software Dirigido por Modelos, JISBD (2010)
9. Eclipse Process Framework Project (EPF), `http://www.eclipse.org/epf/`
10. Ferguson, R.I., Parrington, N.F., Dunne, P., Hardy, C., Archibald, J.M., Thompson, J.B.: MetaMOOSE - an object-oriented framework for the construction of CASE tools. Information and Software Technology 42, 115–128 (2000)
11. Firesmith, D.G., Henderson-Sellers, B.: The OPEN Process Framework. An Introduction, p. 330. Addison-Wesley, London (2002)

12. Grundy, J.C., Venable, J.R.: Towards an Integrated Environment for Method Engineering. In: Proceedings of the IFIP 8.1/8.2 Working Conference on Method Engineering, pp. 45–62. Hall (1996)
13. Harmsen, A.F.: Situational Method Engineering. Moret Ernst & Young (1997)
14. Harmsen, F., Brinkkemper, S.: Design and Implementation of a Method Base Management System for a Situational CASE Environment. In: Asia-Pacific Software Engineering Conference, p. 430. IEEE Computer Society, Los Alamitos (1995)
15. Henderson-Sellers, B.: Method Engineering for OO Systems Development. Communications of the ACM 46(10), 73–78 (2003)
16. Henderson-Sellers, B., Gonzalez-Perez, C., Ralyté, J.: Comparison of Method Chunks and Method Fragments for Situational Method Engineering. In: Proceedings of the 19th Australian Conference on Software Engineering, pp. 479–488. IEEE Computer Society, Los Alamitos (2008)
17. Henderson-Sellers, B., Ralyté, J.: Situational Method Engineering: State-of-the-Art Review. Journal of Universal Computer Science 16, 424–478 (2010)
18. Heym, M., Osterle, H.: A Semantic Data Model for Methodology Engineering. In: Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering, pp. 142–155. IEEE Computer Society Press, Washington, D.C (1992)
19. Kelly, S., Lyytinene, K., Rossi, M.: MetaEdit+ A Fully Configurable Multi User and MultiTool CASE and CAME Environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996)
20. Mirbel, I., Ralyté, J.: Situational method engineering: combining assembly-based and roadmap-driven approaches. Requirements Engineering 11, 58–78 (2006)
21. MOSKitt, `http://www.moskitt.org/`
22. Niknafs, A., Asadi, M.: Towards a Process Modeling Language for Method Engineering Support. In: CSIE 2009: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, pp. 674–681. IEEE Computer Society, Los Alamitos (2009)
23. Ralyté, J., Rolland, C.: An Approach for Method Reengineering. In: Kunii, H.S., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 471–484. Springer, Heidelberg (2001)
24. Ralyté, J., Deneckère, R., Rolland, C.: Towards a generic model for situational method engineering. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 95–110. Springer, Heidelberg (2003)
25. Ralyté, J.: Towards Situational Methods for Information Systems Development: Engineering Reusable Method Chunks. In: Proceedings of the International Conference on Information Systems Development, Vilnius Technika, pp. 271–282 (2004)
26. Reusable Asset Specification (RAS) OMG Available Specification version 2.2. OMG Document Number: formal/2005-11-02
27. Rolland, C., Prakash, N., Benjamen, A.: A Multi-Model View of Process Modelling. Requirements Engineering Journal 4(4), 169–187 (1999)
28. Roger, J.E., Suttenbach, R., Ebert, J., Süttenbach, R., Uhe, I., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE, pp. 203–216. Springer, Heidelberg (1997)
29. Saeki, M.: CAME: The first step to automated method engineering. In: OOPSLA 2003: Workshop on Process Engineering for Object-Oriented and Component-Based Development, pp. 7–18 (2003)
30. Software Process Engineering Meta-model (SPEM) OMG Available Specification version 2.0. OMG Document Number: formal/2008-04-01
31. Xpand, `http://www.eclipse.org/modeling/m2t/?project=xpand`