UNIVERSIDAD POLITECNICA DE VALENCIA

PROS
Centro de Investigación en Métodos de Producción de Software

Technical Report




A Methodological Framework to support Model Driven Method Engineering

Mario Cervera, Manoli Albert, Victoria Torres, Vicente Pelechano





Ref. #:	PROS-TR-2011-14				
Title:	A Methodological Framework to support Model Driven Method Engineering				
Author (s):	Mario Cervera, Manoli Albert, Victoria Torres, Vicente Pelechano				
Corresponding author (s):	Mario Cervera, mcervera@pros.upv.es Manoli Albert, malbert@pros.upv.es Victoria Torres, vtorres@pros.upv.es Vicente Pelechano, pele@pros.upv.es				
Document version number:	1.0	Final version:	Yes	Pages:	21
Release date:	July 2011				
Key words:	Method Fragment, Method Base, Model Transformation, CASE tool				

Chapter 3
A Methodological Framework to support Model Driven Method Engineering

Since the advent of Method Engineering many authors have proposed different approaches to tackle the design and implementation of software production methods. The problem with these approaches is that most of them only focus on one of these tasks, making hard the achievement of the Method Engineering as a whole. On the one hand, the proposals that mainly concentrate on the method design (e.g. [Brinkkemper98, HendersonSellers03, Ralyté03, Mirbel06]) provide rich ways to design methods, but limited (or not-existent) CASE tool generation capabilities. On the other hand, the proposals that mainly focus on the method implementation (e.g. [Kelly96, Grundy96, Roger97, Ferguson00]) provide efficient alternatives to customize CASE tools, but lack the possibility to design software production methods. Unlike these approaches, the methodological framework proposed in this chapter equally encompasses the method design and the method implementation phases of the Method Engineering lifecycle. In order to support these phases in an effective manner, the methodological framework is based on an MDD infrastructure [Atkinson03]. This infrastructure formalizes in a meta-model the concepts that are available for defining methods and provides model transformation techniques to support the definition of mappings from method specifications to the CASE tools that support them.

This chapter has been structured as follows: first, section 3.1 gives an overview of the methodological framework. Then, section 3.2 presents the framework in detail (the MDD infrastructure the framework is built upon, and the framework phases). Finally, section 3.3 concludes the chapter.

3.1. Methodological Framework Overview

This section provides an overview of the various phases that compose the methodological framework introduced in this chapter. These phases are the *method design*, the *method configuration* and the *method implementation* (see Fig. 3.1). In this methodological approach a combination of the assembly and paradigm-based approaches presented in chapter 2 (section 2.1) has been adopted to face the definition of methods. Specifically, this definition is carried out by means of the SPEM standard [SPEM], which is also described in chapter 2 (section 2.2.6).

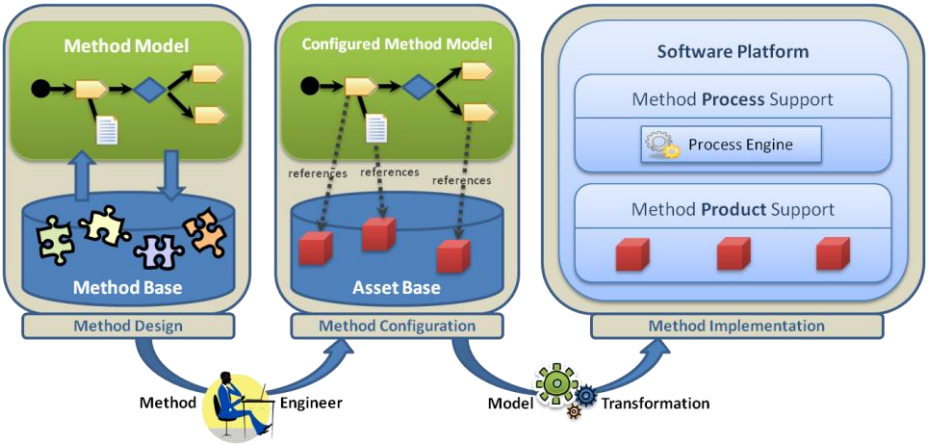

Fig. 3.1. Overview of the methodological framework

Method design

During this phase, the method engineer builds the model of the method using SPEM. This model is composed of two parts: the *product part* and the *process part*[1]. The *product part* represents the artifacts that developers should construct during the execution of a project, and the *process part* represents the procedures that developers must follow to construct such products. The construction of the method model can be performed from scratch or reusing method fragments stored in a Method Base repository that is implemented

[1] The people dimension can be also specified by means of the SPEM primitives: *Role* and *RoleSet*.

following the RAS standard [RAS]. Specifically, the model resulting from this phase constitutes a first version of the method that includes the elements that compose the method (tasks, products, roles, guides, subprocesses, etc.) but no details about the technologies and notations that will be used during its execution are specified. 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, transformations, etc. that are stored as reusable assets[2] in a repository called Asset Base (also implemented following the RAS standard). These assets determine how the method elements will be managed in the final tool. For instance, the product "Business Process Model" can be associated with a "BPMN graphical editor". Thus, the method engineer is indicating that this editor must be included in the generated CASE tool, so that it enables the creation and manipulation of this particular product.

The main benefit of separating the construction of the method model in two phases (i.e. the method design and the method configuration) is that it stresses the importance of reusability, since generic definitions of methods can be stored and then perform different method configurations according to each particular target project or team.

Method implementation

During this phase, the method model is used as input of a model transformation that generates the CASE tool support. This tool provides support to both the *product* and *process* parts of the method. On the one hand, the product support consists of the tools that enable the creation/manipulation of the method products (i.e. the reusable assets associated to the method tasks and products in the previous phase). On the other hand, the process support consists of a process engine that enables the method process execution.

---

[2] These assets represent the tool dimension of the method.

3.2. Methodological Framework

This section presents in detail the methodological framework. First, it details the framework MDD infrastructure, and then each of the framework phases.

3.2.1. Foundations

This section details the MDD infrastructure that lays the foundations of the methodological framework. In particular, this infrastructure is based on meta-modeling and model transformation techniques that allow method engineers to perform the design and implementation of methods.

Meta-modeling

Meta-modeling has always played a key role in the Method Engineering field as it allows the definition at a high level of abstraction of the concepts, constraints and rules that are applicable in the method definition.

The use of meta-modeling in Method Engineering has already been discussed in other works such as [HendersonSellers02], [HendersonSellers06] and [GonzalezPerez08]. In general, proposals that focus on the method design phase usually use meta-modeling as their underlying technique to define the method specifications [Brinkkemper99, Heym92, Mirbel06]. On the other hand, proposals that focus on the method implementation use these techniques to specify the design notations that are to be supported by the generated tools [Grundy96, Kelly96, Roger97].

In the proposal presented in this thesis, meta-modeling techniques are also used for the creation of the method model, in particular following the SPEM standard. A study about the applicability of SPEM to Method Engineering is presented in [Niknafs09]. In this work, the authors present some of the SPEM advantages and disadvantages for supporting the method design. Among the SPEM advantages, in this work are of special interest: (1) wide acceptance in the field of process engineering, (2) good Method Engineering process coverage, (3) support to both product and process parts of methods, and (4) good abstraction and modularization of processes. Regarding its disadvantages, [Niknafs09] 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 this proposal, below the structure of the method fragments from which SPEM models can be assembled is presented in detail. In general, in the Method Engineering 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 [Firesmith02] method fragments are generated by instantiation from one of the top levels classes: Producer, Work Product and Work Unit [HendersonSellers10]. 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

The term method fragment is used in this work to denote the atomic element from which methods can be assembled. Specifically, two different types of method fragments are considered: *product fragments* and *process fragments*. This differentiation offers several advantages, such as (1) leveraging the separation between product and process specification provided by SPEM[3]. Furthermore, it provides the possibility (2) to relate one process fragment with many product fragments and (3) to reuse one product fragment in the definition of many process fragments.

Attending to the different phases identified in the methodological framework, a third type of fragment is actually used. This fragment is called *technical fragment*, term that was first proposed in [Harmsen97]. Specifically, 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 (i.e. they correspond to the reusable assets of the Asset Base).

---

[3] In order to use the same terminology as the used in the Method Engineering field, in this work the product-process separation of methods and the SPEM separation between method content and method process are considered analogous.

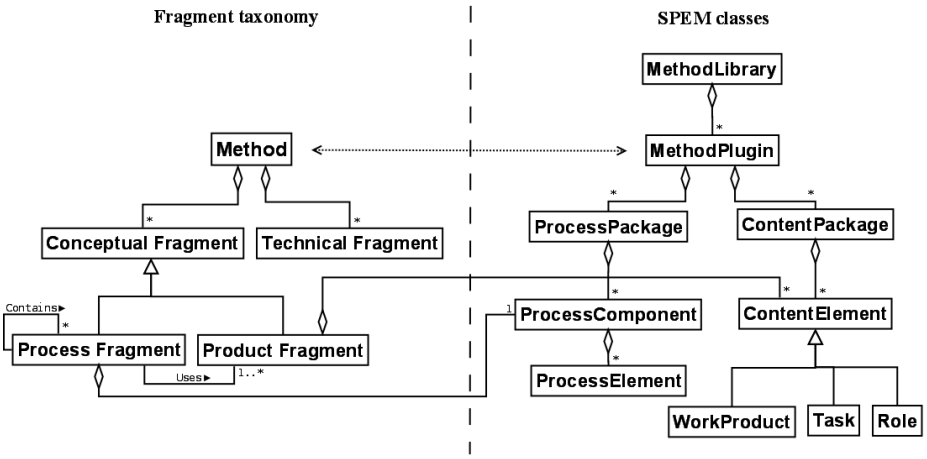

Fig. 3.2. Relationship between method fragments and SPEM classes

In order to illustrate the hierarchical organization of the various types of fragments, the left side of Fig. 3.2 graphically presents the fragment taxonomy. In this taxonomy, the new abstract category *conceptual fragment* (also proposed in [Harmsen97]) is introduced for grouping product and process fragments. Moreover, additional information has been included, e.g. the relationship "contains" between process fragments represents the fact that SPEM processes can contain nested subprocesses, and the relationship "Uses" that one process fragment can reference from one to many product fragments.

On the other hand, the right side of Fig. 3.2 shows a very simplified view of the SPEM meta-model. In SPEM, a method is represented by a *MethodPlugin*, which contains both *ContentPackages* and *ProcessPackages*. Within content packages, *Tasks*, *Roles* and *WorkProducts* are stored. Within process packages processes are stored as instances of the class *ProcessComponent*.

Note that some of these SPEM concepts have been associated with fragments of the taxonomy. These associations illustrate a containment relationship. For instance, process fragments are associated with one *ProcessComponent*. This represents 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. 3.2, method fragments are defined by a series of properties that enable their later retrieval from the repository. These properties are stored in the manifest file of the RAS asset that embodies the fragment. Specifically, some of the properties defined in [Ralyté01] have been used. According to these properties, our method fragments are characterized by:

- *Descriptor*: it contains general knowledge about the fragment. For now, it is composed of 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.
- *Interface*: it describes the context in which the fragment can be reused. For now, it is only composed of the attribute *situation*.

Model transformations

In the previous subsection we showed that the application of meta-modeling in the Method Engineering field is not new. However, the Method Engineering approaches that make use of these techniques do not really take advantage of the possibilities that MDD offers. As stated in [Atkinson03], "the application of MDD techniques improves developers' short-term productivity by increasing the value of primary software artifacts (i.e. the models) in terms of how much functionality it delivers". Following this statement and contrary to what current Method Engineering approaches do, the framework presented in this chapter leverages models going one step further. Defining the method as a model and considering this model as a software artifact permits to face the implementation of the generation of CASE tools by means of model transformations.

In particular, these transformations have been implemented in the CAME environment that supports the proposal as a single M2T transformation using the XPand language [Xpand], which is the language used within the context of the MOSKitt project. Further details about this transformation are provided in the end of section 3.2.2 (method implementation) and chapter 4.

3.2.2. Phases

This section details the phases in which the methodological framework has been divided. As illustrated in section 3.1, these are the *method design*, *method configuration* and *method implementation*.

Method design

During the method design the method model is built using the SPEM standard. The construction of this model is performed by means of a combination of two of the approaches proposed in [Ralyté03]: (1) the paradigm-based and (2) the assembly-based. In order to illustrate how these approaches are applied in the framework, the *Map* process meta-model proposed in [Rolland99] is used.

The paradigm-based approach

Fig. 3.3 shows how the method model is built 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, in this proposal method models are built by instantiating a meta-model (i.e. the SPEM meta-model).

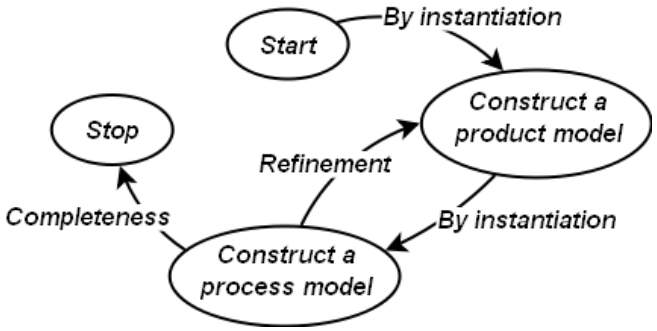

Fig 3.3. Paradigm-based approach (adapted from [Ralyté03])

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. 3.4 shows how the assembly-based approach is carried out. This process is followed when the method engineer wants to reuse product or process fragments stored in the Method Base during the construction of the method model.

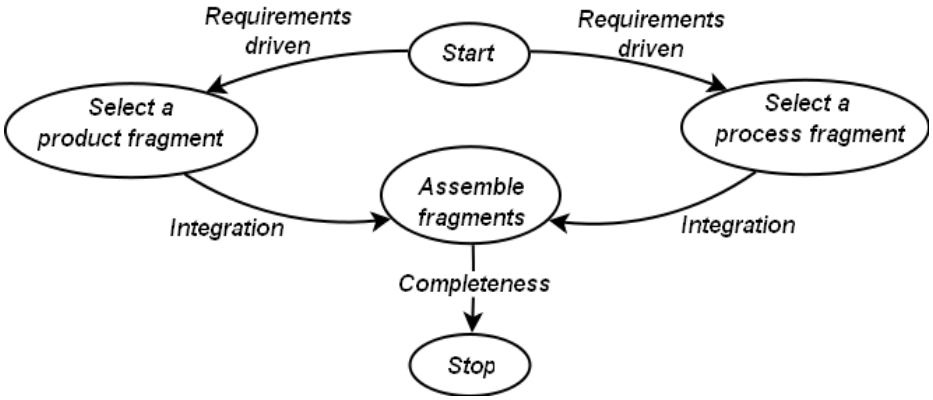

Fig. 3.4. Assembly-based approach (adapted from [Ralyté03])

As shown in the figure, the fragment selection is requirements driven. Thus, 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 3.2.1). 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 c*ontent package*. For process fragments, the process elements are included as a subprocess in the method under construction. For this purpose, SPEM provides the class *CapabilityPattern*.

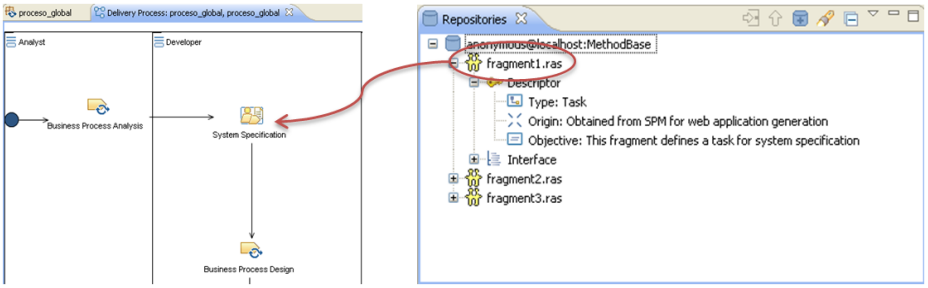

Fig. 3.5. Example of method fragment integration

Fig. 3.5 shows an example of integration of a method fragment into a method model, which has been created by means of the EPF Composer Editor (a SPEM editor provided in the EPF Project [EPF]). The right side of this figure shows an Eclipse view implementing a repository client. Its content represents method fragments that are stored in the Method Base. Through this view, the method engineer can search and select method fragments and integrate them into the method model.

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. 3.6 shows the process that must be followed.

---

[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 Fig. 3.4.

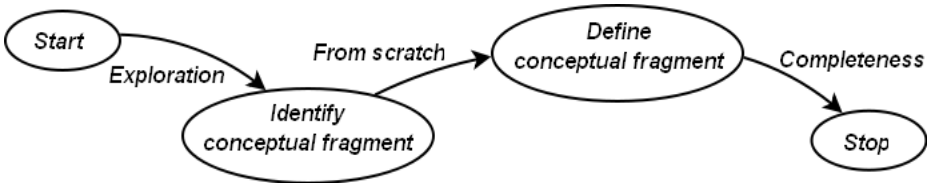

Fig. 3.6. Conceptual fragment creation (adapted from [Ralyté04])

First, the method engineer explores the method model in order to identify the elements that must be included in the conceptual fragment to be created. 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 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. 3.7 shows how the method configuration 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, the method engineer associates it with a task or product of the method model.

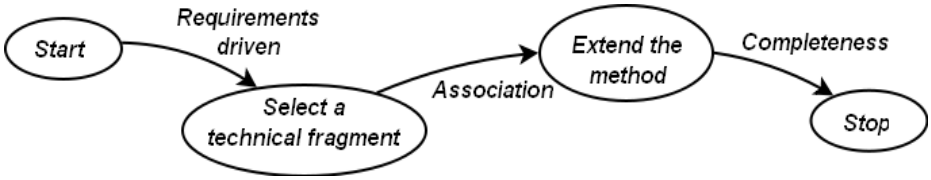

Fig. 3.7. Process model for technical fragment association

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. 3.6 is followed. First, the required tool is implemented ad-hoc for the method under construction. For instance, in the CAME environment that supports this proposal (see chapter 4) 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.

Below, the various types of technical fragments that can be stored in the Asset Base are detailed. Furthermore, for each of these types, it is specified 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 CASE tools for their manipulation (e.g. the "BPMN meta-model" can be associated 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 CASE tools for their manipulation (e.g. a "BPMN graphical editor" can be associated to the product "Business Process Model").
- *Transformation*: model transformations can be associated to tasks of the method. This entails that these tasks will be automatically executed in the generated CASE tool by means of the model transformations (e.g. a M2T transformation can be associated 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 generated CASE 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.

Fig. 3.8 shows an example of a technical fragment containing a BPMN graphical editor. This fragment is packaged following the RAS standard. According to RAS, reusable assets are represented by zip files that contain a manifest describing the asset properties and one or more artifacts that compose the asset. Specifically, the asset of Fig. 3.8 is composed of the manifest file and the Eclipse plugins that implement the graphical editor.

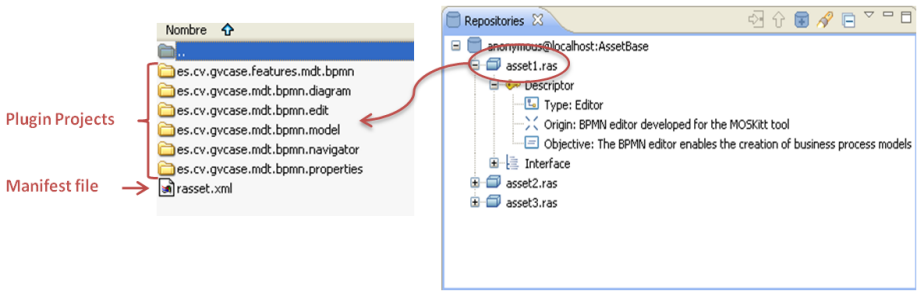

Fig. 3.8. Example of technical fragment: a BPMN editor

Method implementation

This section describes the part of the methodological framework that deals with the construction of the CASE tool that supports the method resulting from the method configuration phase. Specifically, this tool is generated by means of model transformations. Fig. 3.9 provides a graphical overview of this process and Fig 3.10 a detailed view of the structure of the generated CASE tools.

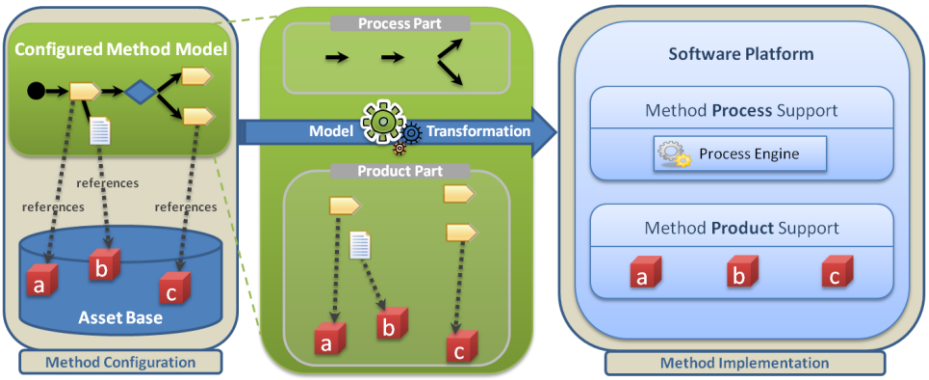

Fig. 3.9. Overview of the tool generation process

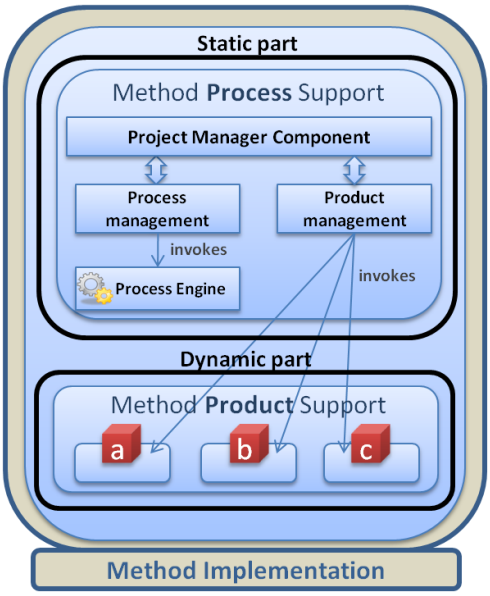

Fig. 3.10. Structure of the generated CASE tools

The core of the generation process is a model transformation that obtains a software tool supporting the method specified in the configured method model. As shown in Fig. 3.9, the transformation uses the product and process parts of the method model in order to obtain a CASE tool that gives support to both parts as follows:

- The support provided for the *product part* involves all the resources that enable the manipulation of the method products. This support is given by the software components that make up the infrastructure of the tool and correspond to the technical fragments that were associated to the elements of the method during the method configuration phase.
- The support provided for the *process part* corresponds to a software component (i.e. the Project Manager Component) that enables the execution of method instances by means of a process engine. During the method execution, this component invokes the different software resources that allow the software engineers to create and manipulate the method products.

Software support for the product part

This subsection focuses on the part of the model transformation that obtains the tool support for the product part of the method. This product support constitutes the dynamic part of the tool, i.e. the part that is obtained from the method model and thus it is dependent of the method that has been specified. Specifically, it refers to the tools (editors, transformations, etc.) that have to be integrated into the final tool to enable the creation and manipulation of the method products. For instance, a method that includes a product such as a "Business Process Model" requires the inclusion within the CASE tool of a proper editor to manage this kind of models.

Furthermore, to obtain a valid product support it is necessary to solve the dependencies of the software components required to support the product part with other software components. Therefore, two steps must be performed by the model transformation: (1) identifying the software resources necessary to support the tasks and products of the method and (2) solving the dependences between software resources.

In a first step, the model transformation explores the method model and identifies the software resources that are necessary to support the tasks and products of the method. The software resources are identified by means of the reusable assets (technical fragments) that were associated to these elements during the method configuration. Note that when a task or a product does not have an associated asset, the generated tool will not provide support to that element.

Once the required software resources are identified, it is necessary to solve the potential conflicts that can arise when integrating these resources into the same platform. To achieve this goal, the dependencies between software resources are specified within the assets. This specification allows the transformation to retrieve the dependencies for each software resource identified in the previous step and to include them in the final tool. Note that, for this purpose, the resources that represent the dependencies must also be stored in the Asset Base repository.

As an example consider the asset of Fig. 3.8 containing the MOSKitt BPMN editor. This asset defines a dependency with the MOSKitt MDT component[5]. Therefore this component must also be included in the final tool.

Software support for the process part

In addition to the support provided for the product part of the method, the generated tool also provides support for the process part. The process support is provided by means of a software component that is always included in the generated tools. This component is called the *Project Manager Component* and constitutes the static part of the tool (i.e. its implementation is independent of the method that has been specified). This component implements a graphical user interface (GUI) that guides and assists software engineers during the execution of method instances (projects). To make this possible, the Project Manager Component uses the configured method model at runtime[6].

Specifically, the Project Manager Component is divided into four components of a lower level of granularity. These components are the following (see Fig. 3.11):

- **Project Manager (PM)**. This is the core component as it centralizes the management of the other three subcomponents. In addition, it contains the implementation of the GUI.
- **Process Management**. This component makes the access to the process engine transparent for the PM. Note that SPEM does not contain executable semantics. Therefore, up to now the process engine is implemented as a light-weight process engine that keeps the state of the running method instances. As future work, the integration of the Activiti engine [Activiti] into the CAME tool that support the proposal is being planned. This will require the definition of a model transformation to map SPEM models into BPMN 2.0 models that can be executed by Activiti.

---

[5] The MOSKitt MDT component implements the functionality that is common to all the MOSKitt graphical editors (such as copy & paste, view creation, etc.)

[6] Runtime in this context corresponds to the method instances execution in the CASE tool

- **Product Management**. This component is in charge of invoking the tools that support the product part of the method based on the state of the running method instance. In other words, it invokes the editor, transformation, etc. that is needed to perform the current task of the method.
- **Method Specification**. This component loads the different elements of the method model (roles, tasks, products, etc.) to facilitate later access to them.

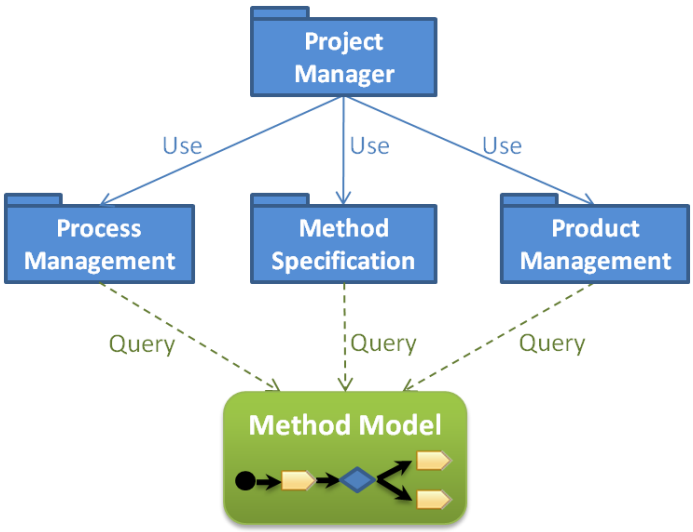

Fig. 3.11. Structure of the Project Manager Component

3.3. Conclusions

The combination of the MDD paradigm and the technology provided by the MOSKitt platform represents an adequate setting to turn Method Engineering into reality. Our methodological framework benefits from this combination. On the one hand, the application of MDD techniques has enabled the coverage of both the design and implementation of software production methods. On the other hand, the MOSKitt plug-in based architecture and its integrated modeling tools provide a suitable platform to support the framework and does not present the deficiencies found in current tools.

Specifically, this chapter has focused on the most theoretical part of this methodological framework.

Our framework aims to provide assistance to method engineers during the definition of project-specific methods and the construction of the corresponding supporting tools. Following the MDD paradigm, meta-modeling techniques based on the SPEM standard are used for building the method specifications as machine-processable models. One of the main novelties of the framework is that, unlike current Method Engineering approaches, it leverages these models by using them as inputs of model transformations that perform the CASE tool generation process.

References

[Activiti] Activiti, http://www.activiti.org/

[Atkinson03] Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software, IEEE Computer Society, 20, 36-41 (2003)

[Brinkkemper98] Brinkkemper, S., Saeki, M., Harmsen, F.: Assembly Techniques for Method Engineering. CAiSE '98: Proceedings of the 10th International Conference on Advanced Information Systems Engineering, Springer-Verlag, 381-400 (1998)

[Brinkkemper99] Brinkkemper, S.; Saeki, M., Harmsen, F.: Meta-Modelling Based Assembly Techniques for Situational Method Engineering. Information Systems, 24, 209-228 (1999)

[EPF] Eclipse Process Framework Project (EPF), http://www.eclipse.org/epf/

[Ferguson00] 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)

[Firesmith02] Firesmith, D.G., Henderson-Sellers, B.: The OPEN Process Framework. An Introduction. Addison-Wesley, London, UK, 330pp (2002)

[GonzalezPerez08] Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley Publishing (2008)

[Grundy96] 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, Hall, 45-62 (1996)

[Harmsen97] Harmsen, A.F.: Situational Method Engineering. Moret Ernst & Young (1997)

[HendersonSellers02] Henderson-Sellers, B., Lowe, D., Haire, B.: OPEN Process Support for Web Development. Annals of Software Engineering 13, 163-201 (2002)

[HendersonSellers03] Henderson-Sellers, B.: Method Engineering for OO Systems Development. Communications of the ACM Vol. 46. Nº 10, pp. 73-78, (2003)

[HendersonSellers06] Henderson-Sellers, B.: Method Engineering: Theory and Practice. In Information Systems Technology and Its Applications. 5th International Conference ISTA'06. Klagenfurt, Austria, D. Karagiannis, H.C. Mayr, Eds. Lecture Notes in Informatics (LNI) – Proceedings, Volume P-84, Gesellschaft Für Informatik, Bonn, 13- 23 (2006)

[HendersonSellers10] Henderson-Sellers, B., Ralyté, J.: Situational Method Engineering: State-of-the-Art Review. Journal of Universal Computer Science, 16, 424-478 (2010)

[Heym92] Heym, M., Österle, H.: A Semantic Data Model for Methodology Engineering. In: 5th Workshop on Computer-Aided Software Engineering, pp. 142-155. IEEE Press, Los Alamitos (1992)

[Kelly96] Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. CAiSE, 1-21 (1996)

[Mirbel06] Mirbel, I., Ralyté, J.: Situational method engineering: combining assembly-based and roadmap-driven approaches. Requirements Engineering, 11, 58-78, (2006)

[Niknafs09] Niknafs, A., Asadi, M.: Towards a Process Modeling Language for Method Engineering Support. CSIE'09: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, IEEE Computer Society, 674-681 (2009)

[Ralyté01] Ralyté, J., Rolland, C.: An Approach for Method Reengineering. ER'01: Proceedings of the 20th International Conference on Conceptual Modeling, Springer-Verlag, 471-484 (2001)

[Ralyté03] Ralyté, J., Deneckère, R., Rolland, C.: Towards a generic model for situational method engineering. CAiSE'03: Proceedings of the 15th international conference on Advanced information systems engineering, Springer-Verlag, 95-110 (2003)

[Ralyté04] 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, 271-282 (2004)

[RAS] Reusable Asset Specification (RAS) OMG Available Specification version 2.2. OMG Document Number: formal/2005-11-02

[Roger97] Roger, J. E., Suttenbach, R., Ebert, J., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Springer , 203-216 (1997)

[Rolland99] Rolland, C., Prakash, N., Benjamen, A.: A Multi-Model View of Process Modelling. Requirements Engineering Journal 4(4), 169-187 (1999)

[SPEM] Software & Systems Process Engineering Meta-model (SPEM) OMG Available Specification version 2.0. OMG Document Number: formal/2008-04-01

[Xpand] Xpand, http://www.eclipse.org/modeling/m2t/?project=xpand