A Technological Framework to support Model Driven Method Engineering[1]

Mario Cervera[1], Manoli Albert[1], Victoria Torres[1], Vicente Pelechano[1], Javier Cano[2], Begoña Bonet[3]

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

[2]Prodevelop S.L.
46001 Valencia, Spain
fjcano@prodevelop.es

[3]Conselleria de Infraestructuras y Transporte, Generalitat Valenciana, 46010 Valencia, Spain
bonet_beg@gva.es

Abstract

Over the last two decades many approaches have contributed to establish a solid theoretical basis in the area of Method Engineering, but very few engineering tools have been developed to provide software support to their research results. This situation is mainly due to the complexity of developing Computer-Aided Method Engineering environments that enable the specification of Software Production Methods (SPM) and the construction of CASE tools to support them. In order to reduce this complexity, we advocate for the use of the MDD paradigm, which promotes the use of models as the primary artifact in the development process. Following this paradigm, in this paper we present a Model Driven Method Engineering approach to perform the automatic construction of tools that support SPMs by means of model transformations. This work is contextualized within a more challenging proposal that provides a methodological framework and a software infrastructure for the construction of SPM, covering from their specification to the construction of the tool support.

Keywords

Method Engineering, Model Driven Development, CAME environment, CASE tool generation.

1. Introduction

A Software Production Method (SPM) is an integrated set of activities, roles, products, guides and tools for providing efficient and effective support in the software development process. In the Software Engineering (SE) field, CASE environments provide software support to SPMs contributing to improve the software development process in terms of productivity, maintainability, reusability and quality of the developed software. However, despite the benefits that the use of CASE tools provides, these are not used as widely as expected. One of the reasons for this is that CASE tools are implemented to give support to a single SPM, paying no attention to the flexibility required by real software projects. As a result, developers find difficult to work with such tools as they do not allow them to adapt the SPM to the requirements of a specific project [21].

One way to overcome this problem is by reconsidering the way in which these tools are built. The construction of such tools is one of the main concerns of the Method Engineering (ME) discipline. ME is defined as *the engineering discipline to design, construct and adapt methods, techniques and tools for the development of IS* [2]. Within the ME field, Computer Aided Method Engineering (CAME) environments enable the construction of SPMs and the software tools that support them. However, providing such support is

[1]This work has been cofinanced by the Conselleria de Infraestructuras y Transporte by means of the Fondo Europeo de Desarrollo Regional (FEDER) and the Programa Operativo de la Comunitat Valenciana 2007-2013.

not an easy task being a clear example the low implementation degree and deficiencies found in existing CAME environments [17].

To improve this situation, in this work we advocate for the use of the MDD paradigm, which proposes using models as the primary artifact of the development process [1], in the ME field. Thus, this work provides a Model Driven Method Engineering approach to perform the construction of tools that support SPMs by means of model transformations. The work is being developed as part of a more challenging proposal [4]. This proposal contributes to the ME area by providing a methodological framework and a software infrastructure for the construction of SPMs. The methodological framework covers from the specification of the SPM to the construction of the tool support. The present work focuses on the last phase of this proposal where software tools are built from SPM specifications.

The remainder of the paper is structured as follows. In section 2 we briefly present the state of the art focusing on the limitations of the existing CAME environments. Then, in section 3 we provide a brief overview of the ME proposal in which this work is contextualized. Section 4 presents the strategy designed to automatically obtain software tools to support specific SPMs. Finally, section 5 draws some conclusions and further work.

2. State of the art

The first research work in the ME field was developed in the early nineties by Kumar and Welke who established the basis of this area [14]. Later, these foundations have been consolidated with several proposals such as Brinkemmper's [2] and Hofstede's [12]. Since then, different proposals try to provide an answer to the existing problems in this area. This is the case of proposals such as Ralyté's [15, 19], Henderson-Sellers' [10], Prakash's [18] or Harmsen's [9] which tackle the method construction by assembling pieces or fragments, proposing techniques for the efficient selection and assembly of these pieces. These proposals have contributed to establish a solid theoretical basis for the ME area. However, the existing tool support for this basis does not live up to the expectations due to the complexity of putting this theory into practice. This problem becomes evident in [17] where a study of different CAME environments is presented. This study concludes that existent environments are incomplete prototypes that only cover part of the ME process. This is one of the reasons why these tools have not achieved the expected industrial success and just MetaEdit+ [13] has been commercialized. Examples of these CAME tools are MERU, which supports Prakash's and Gupta's proposals [7], DECAMERONE, which supports Brinkkemper's [3], MENTOR [22], MERET [11] or KOGGE [21].

These CAME environments, in general, present important deficiencies. Between these deficiencies we highlight: (1) lack of support to the definition of SPMs and (2) lack of support to the automatic generation of CASE tools from the SPM definitions. This situation points out that there is an actual need for tools that provide better support to ME. The problem is the high complexity that entails the construction of these tools as they must provide support both to the SPM specification and the CASE tool generation. In order to overcome this problem some approaches apply the MDD paradigm using metamodelling languages either to define design notations [6] or SPM specifications [11]. However, we find that these approaches do not really take advantage of the possibilities that the MDD techniques offer. As stated in [1], "the application of MDD techniques improves developers' short-term productivity by increasing the value of primary software artifacts (e.g. the models) in terms of how much functionality it delivers". Following this statement and contrary to what current ME approaches do, we want to leverage models going one step further. Defining the SPM as a model and considering this model as a software artifact allows us to face the implementation of the generation of software support tools by means of model transformations. The use of model transformations as the means to carry out the tool generation is the main concern of this paper and is thoroughly detailed in section 4.

3. ME proposal overview

In order to put into context the work presented in this paper, this section briefly introduces the proposal presented in [4]. This proposal covers

different stages of the ME lifecycle, in particular from the specification of the SPM to its implementation (where the tool that supports the SPM is built). Figure 1 presents a graphical overview of the proposal. Each of its phases is detailed in the next subsections.

3.1. Method design

During this phase, the method engineer builds the *Method Model* by identifying all the elements involved in the SPM. The most significant elements used in the *Method Model* construction are the following:

- **Task:** It represents an activity performed during the execution of a SPM instance (e.g. *business process analysis, web specification,* etc.).
- **Product:** It represents an artifact that is either consumed or generated in a task (e.g. *business process model, structural model,* etc.).
- **Role:** It represents an agent that participates in a SPM performing different tasks. This can refer to a human being agent (e.g. *analyst, developer,* etc.) or to an automated system.
- **Flow Connector:** It represents the order in which two associated tasks (each one in a different end) are executed.
- **Gateways:** It represents points within the SPM where the flow is diverged or converged depending on the gateway type.
- **Guide:** It is a document that provides some assistance to perform a task or to manipulate a specific product.

We distinguish two parts in the *Method Model*, the *product part*, which represents the artifacts that developers should construct during the execution of a SPM project, and the *process part*, which consists of the procedures that developers must follow to construct such products. For the construction of the *Method Model* we provide a *Method Base* repository. The *Method Base* contains method fragments (descriptions of IS engineering methods, or any coherent part thereof [8]) that can be reused in the design of new *Method Models*. It is important to note that the *Method Model* does not contain details about the languages or technologies that are going to be used during the execution of the SPM; this is done in the next phase.

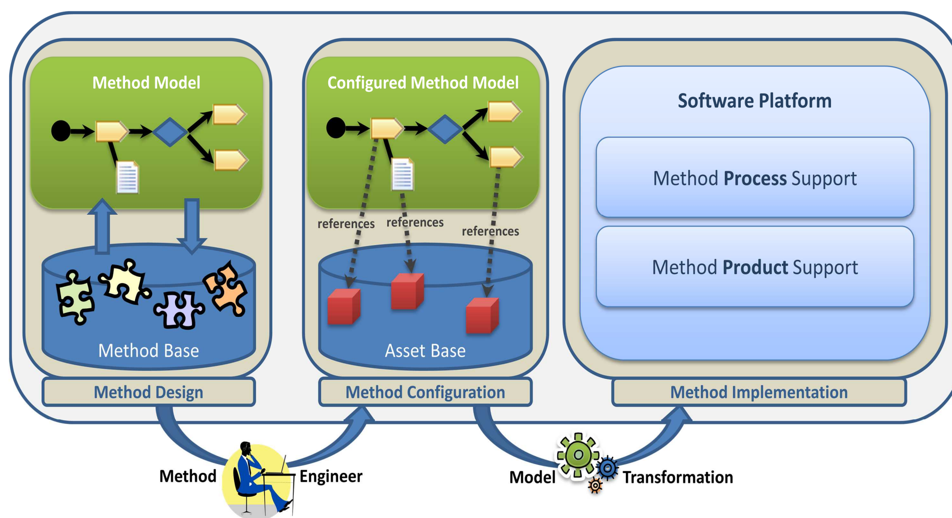

Figure 1. ME proposal overview

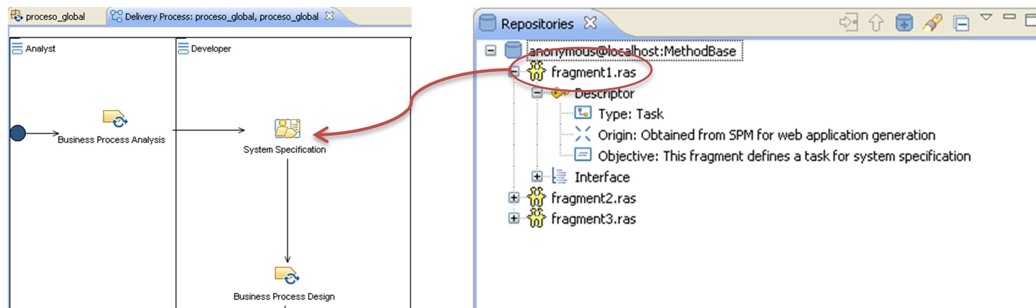

Figure 2. Example of method fragment integration

Figure 2 shows an example of integration of a method fragment into a *Method Model*, which in our proposal is created by means of the EPF Composer Editor, a Software Process Engineering Meta-Model (SPEM) [23] editor provided in the EPF Project [5]. 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* as reusable assets following the RAS (Reusable Asset Specification) standard [20]. Through this view, the method engineer can search for and select method fragments to integrate them into the *Method Model*.

3.2. Method configuration

During this phase, the method engineer associates the elements included in the *Method Model* with metamodels, editors, transformations, etc., which are stored in the *Asset Base* repository. These assets configure the elements of the *Method Model* and determine how they will be managed in the tool built for supporting the SPM. The assets contained in the *Asset Base* can be built either in other SPMs or ad-hoc for the SPM under construction (the method engineer can use the tools provided in our CAME environment for this purpose). Specifically, in our proposal the assets contained in the *Asset Base* correspond either to Eclipse plugin/feature[2] projects that implement editors, metamodels or transformations, or to task guidelines.

The elements of the *Asset Base* are specified following the RAS standard [20]. According to RAS, reusable assets are represented by zip files that contain a manifest describing the asset and one or more artifacts that compose the asset. Figure 3 shows an example of an asset containing a BPMN editor. This asset could be associated, for instance, to a SPM product called "*Business Process Model*" to specify that this product will be managed in the generated tool using a BPMN editor.

At the end of this phase, the *Method Model* has evolved into a new stage where detailed information about the technological support of SPM tasks is given. We call *Configured Method Model* to the model resulting from this phase.

3.3. Method implementation

During this phase a model transformation is executed to automatically obtain the tool that supports the SPM. This transformation takes as input the *Configured Method Model* previously obtained during the *Method Configuration* phase. The details of this phase, which are the focus of this paper, are given in the following subsection.

4. Automatic generation of tools for SPM support

This section describes the part of the ME approach that deals with the construction of the software tool to support the SPM. The construction process is based on the application of the MDD paradigm; so, these software tools are generated from SPM specifications by means of model transformations. Figure 4 provides a graphical overview of this process.

[2] An Eclipse feature is a group of Eclipse plugins.

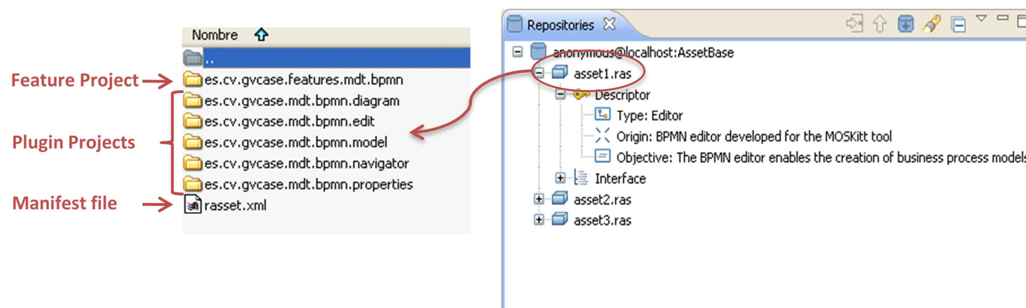

Figure 3. Example of reusable asset: a BPMN editor

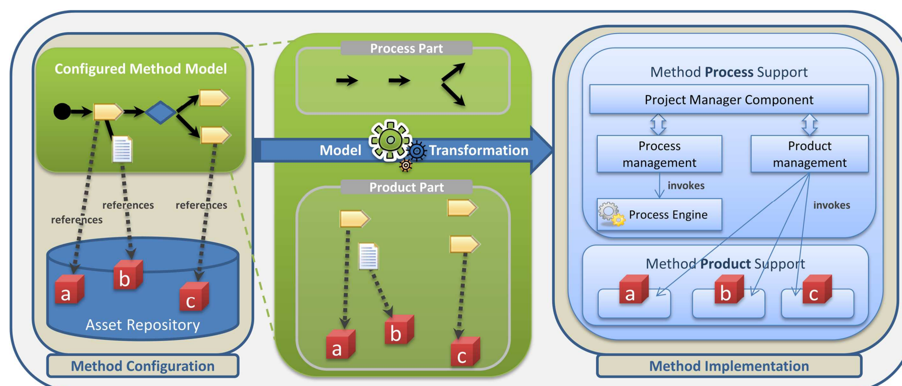

Figure 4. Overview of the tool generation process

The core of the generation process is a model transformation that obtains a software tool supporting the SPM specified in the *Configured Method Model*. As shown in the figure, the transformation uses the product and process parts of the SPM model to give support to both parts as follows:

- The support provided for the *product part* involves providing all the resources that enable the manipulation of the SPM products. This support is given by the software components that make up the infrastructure of the tool and correspond to the assets that were associated to the SPM elements in the *Method Configuration*.
- The support provided for the *process part* corresponds to a new component that enables the execution of SPM instances by means of a process engine. During the SPM execution, this component invokes the different software resources that allow the software engineers to create and manipulate the SPM products.

The generation process of figure 4 has been implemented in the CAME environment developed to support the proposal [4]. In this context, the generated tools are built as Eclipse applications, in particular based on the MOSKitt tool [16]. This means that these tools are built as MOSKitt reconfigurations that only contain the set of plugins that implement the software support required by the SPM.

The use of the MOSKitt platform implies that: (1) the software resources that give support to the product part of the SPM correspond to Eclipse plugins and (2) the final tool is obtained from a *Product Configuration File* (.product file). This type of files gathers all the required information to automatically[3] generate an Eclipse-based tool such as MOSKitt. So, considering that the tool is obtained from a *Product Configuration File,* the

[3] The Eclipse Product Export Wizard (functionality provided in org.eclipse.pde) automatically generates an Eclipse-based application from a .product file.

model transformation is in fact a model-to-text (M2T) transformation implemented using the Xpand language [24]. This transformation takes as input the *Configured Method Model* and generates a .product file through which the final tool will be automatically generated. In order to generate this file, the M2T transformation must identify the software resources (Eclipse plugins) in charge of providing support to the SPM. Once these resources have been identified, the transformation includes in the *Product Configuration File* the list of features that need to be deployed in the final tool (MOSKitt construction is based on features).

More insights on this M2T transformation and the tools obtained for product and process support in the final tool are presented in the next subsections.

4.1. Software support for the product part

This section focuses on the part of the M2T transformation that obtains the tool support for the product part of the SPM. This product support refers to the tools (editors, transformations, etc.) that have to be integrated into the final tool to enable the manipulation of the SPM products and tasks. For instance, a SPM that includes a product such as a "*Business Process Model"* requires the inclusion within the tool supporting the SPM of a proper editor to manage this kind of models.

Furthermore, to obtain a valid product support it is also necessary to solve the dependencies of the software components required to support the SPM product part with other software components. Therefore, we distinguish two steps in the M2T transformation that obtains the product support part: (1) *identifying the software resources necessary to support the tasks and products of the SPM* and (2) *solving the dependences between software resources*.

Identifying software resources

The M2T transformation explores the SPM model and identifies the software resources that are necessary to support the tasks and products of the SPM. The software resources are identified by means of the assets that were associated to these elements during the *Method Configuration* phase. Note that when a task or a product does not have an associated asset, the generated tool will not provide support to that element.

It is also important to highlight that the integration of these resources into the MOSKitt reconfiguration representing the final tool can be automatically performed since these resources correspond to features and plugins created within the Eclipse/MOSKitt platform itself. Thus, the integration of tools developed outside of the context of Eclipse/MOSKitt cannot be guaranteed.

In figure 5 two Xpand rules of the M2T transformation are shown. In these rules the list of features of the *Product Configuration File* is generated. The first rule is invoked for each instance of the class *ContentElement* (i.e. tasks and products). This rule invokes the second rule, which produces the output. The second rule accesses the property "FeatureID" of the content elements. This property is created during the asset association and contains the identifier of the feature (software resource giving support to the content element) packaged in the asset.

```
«DEFINE contentElement FOR uma::ContentElement-»
«EXPAND asset FOREACH this.assets.typeSelect(uma::ReusableAsset)-»
«ENDDEFINE»

«DEFINE asset FOR uma::ReusableAsset-»
«FOREACH this.methodElementProperty AS property-»
«IF property.name == "FeatureID"-»
<feature id="«property.value»" version="0.0.0"/>
«ENDIF-»
«ENDFOREACH-»
«ENDDEFINE»
```

Figure 5. Excerpt of the M2T transformation

Solving dependencies between software resources

Once the required software resources are identified, it is necessary to solve the potential conflicts that can arise when integrating these resources (plugins) into the same platform (MOSKitt). To achieve this goal, we specify the dependencies between software resources 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 *Product Configuration File*.

As an example consider the asset of figure 3 containing the MOSKitt BPMN editor. This asset defines a dependency with the MOSKitt MDT component[4]. Therefore its feature must also be included in the .product file so that the plugins implementing this component are also included in the final tool.

4.2. Software support for the process part

In addition to the support provided for the product part of the SPM, according to our proposal, the generated tool also provides support for the process part. This support guides and assists users during the execution of SPM instances (projects).

The process support is provided by means of a software component (the *Project Manager Component*) that is common to all SPMs. This component implements a graphical user interface (GUI) that enables the execution of SPM instances. To make this possible, the *Project Manager Component* uses the *Configured Method Model* at runtime (runtime in this context corresponds to the SPM instances execution in the CASE tool).

Considering these aspects of the process support, the M2T transformation must always include in the *product configuration file* a pre-defined feature that groups the set of plugins that implement the *Project Manager Component*.

The *Project Manager Component* endows the generated tool with a GUI composed of a set of Eclipse views (see Figure 6[5]). Each of these views provides a specific functionality but their common goal is to facilitate the user participation in a specific project. The details of these views are the following:

- **Product Explorer**: This view shows the set of products that are handled (consumed, modified and/or produced) by the ongoing and finished tasks of the process. This view can be filtered by roles so that users belonging to a specific role have only access to the products they are in charge of. Then, from each product, the user can open the associated editor to visualize or edit its content.
- **Process**: This view shows the tasks that can be executed within the current state of the project. The execution of the tasks can be performed automatically (by launching the transformation associated to the task as a software asset) or manually by the software engineer (by means of the software resource associated to the output product of the task). Similarly to the *Product Explorer*, this view can be filtered by role, showing just the tasks in which the role is involved in.
- **Guides**: This view shows the list of guides associated to the task selected in the *Process* view. The objective of these guides is to assist the user during the execution of such task, providing some insights on how the associated products should be manipulated. These guides correspond to resources that were associated to tasks during the configuration step of the SPM.
- **Product Dependencies**: This view shows the dependencies that exist between the products that are handled in the project. So, it allows users to identify which products cannot be created or manipulated because of a dependent product has not yet been finished. In addition, these dependencies are organized by roles. This organization gives to the user the knowledge of who is responsible of those products he/she is interested in.

---

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

[5] Available also at http://users.dsic.upv.es/~vtorres/moskitt4me/

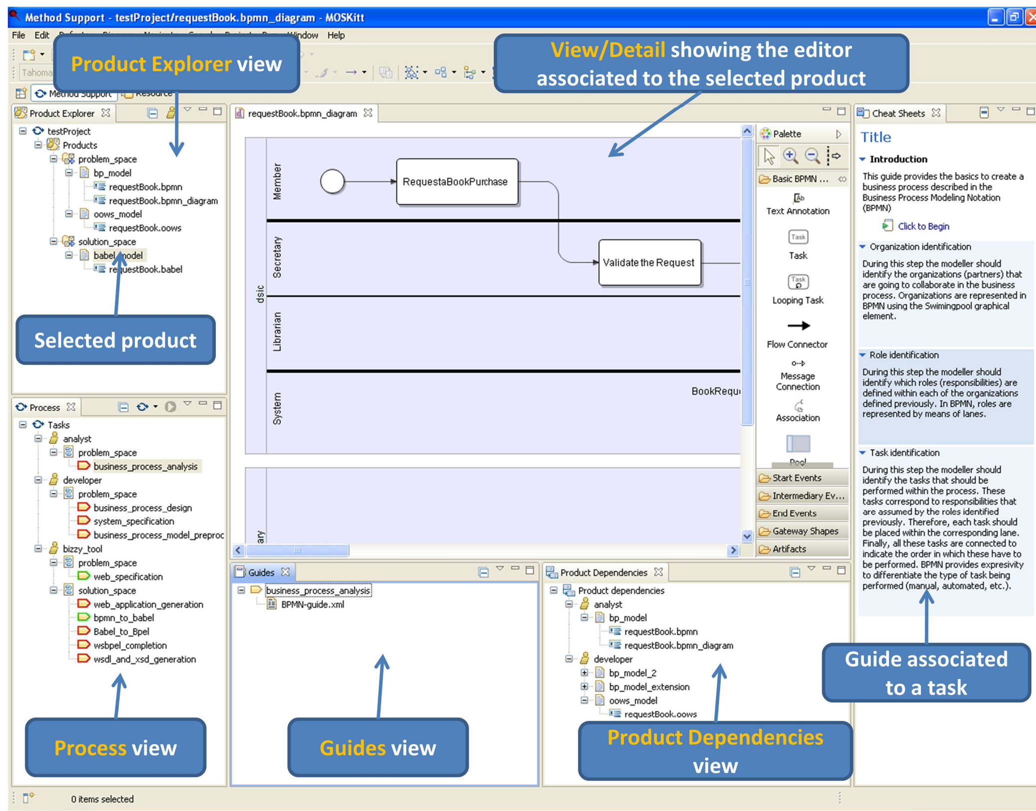

Figure 6. Project Manager GUI

Regarding the implementation of the *Project Manager Component*, it has been divided into four components of a lower level of granularity. The M2T transformation that generates the *product configuration file* always includes a feature that groups the Eclipse plugins that implement these four components. Even though the implementation of these components is independent of the SPM, as stated previously, they need the information stored in the *Configured Method Model* to work properly in the generated tool. Figure 7 depicts graphically these four components.

- **Project Manager**. This is the core component. It implements the GUI of the *Project Manager Component* and gives support to the process part of the final tool. To do so, this component uses the other three.
- **Process Management**. This component implements a light-weight process engine that keeps the state of the running SPM instances. Given a SPM instance it provides a set of methods that return the current tasks and also allow the method engineer to mark them as *completed* in order to enable the progress of the process. Note that, to make this progress possible, the component must access the SPM model and retrieve the distribution of the tasks along the SPM process.
- **Product Management.** This component is in charge of the management of the products and tasks. Regarding products, the component identifies the editor that is required to manipulate such product. Regarding tasks, we differentiate between automated and manual tasks. For automated tasks, the component obtains the transformations that have to be executed. For manual tasks it obtains the editor that allows creating and editing the products manipulated in this task. All this

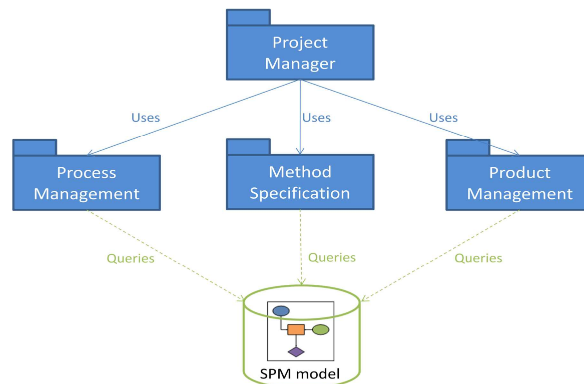

Figure 7. Structure of the Project Manager Component

information is contained in the SPM model, in particular in the assets associated to the tasks and products included in the model. Therefore, this component also needs to access the SPM model to get this information.

- **Method Specification**. This component loads the different elements of the SPM model (roles, tasks, products, etc.) to facilitate later access to them. All these elements are obtained from the SPM model.

5. Conclusions

The development of CAME tools is a task that has proven itself as highly complex. When facing this challenge, the use of techniques that simplify this process becomes crucial. Considering this, some ME approaches have used MDD techniques using metamodelling languages either to define design notations [6] or SPM specifications [11]. The problem is that these approaches fall short when providing a solution to ME as they do not really take advantage of the possibilities that these techniques offer.

Considering this lack, we want to leverage models going one step further. With this purpose we have presented a MDD approach that not only uses models for the specification of SPMs but also uses them as software artifacts, tackling the generation of tools to support them by means of model transformations. In particular, this work is contextualized within a broader proposal [4]. This proposal presents a methodological framework for the construction of SPMs, which covers from the SPM specification to the generation of the tool support. Specifically, this process is divided into two phases, being the last one the central focus of this paper.

Regarding future work, we are working on the improvement of the CAME environment that supports our proposal. We are enhancing: (1) the management of the dependencies between the resources that have to be included in the final tool supporting the SPM under construction, and (2) the workflow engine that enables the execution of the SPM process and gives support to the process part of the proposal.

References

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

[2] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tool. Information and Software Technology 38 (1996)

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

[4] Cervera, M., Albert, M., Torres, V., Pelechano, V.: A Methodological Framework and Software Infrastructure for the Construction of Software Production Methods. International Conference on Software Processes, (2010)

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

[6] 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)

[7] Gupta, D., Prakash, N.: Engineering Methods from Method Requirements Specification. Requirements Engineering, Vol. 6 (2001)

[8] Harmsen, A. F., Arnhem, T., Ernst, M., Consultants, Y. M., Gegevens, C., Bibliotheek, K., Haag, D., Frank, H. A.: SITUATIONAL METHOD ENGINEERING PROEFSCHRIFT 1968.

[9] Harmsen, F., Brinkkemper, S.: Design and Implementation of a Method Base Management System for a Situational CASE Environment. APSEC (1995)

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

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

[12] Hofstede, A., Verhoef, T. F.: On the Feasibility of Situational Method Engineering. Information Systems. 6/7 Vol. 22. (1997)

[13] S. Kelly, K. Lyytinene, M.Rossi. MetaEdit+ A Fully Configurable Multi User and MultiTool CASE and CAME Environment. CAiSE 1996.

[14] Kumar, K., Welke, R. J.: Methodology Engineering: A Proposal for Situation-Specific Methodology Construction. Challenges and Strategies for Research in Systems Development, John Wiley & Sons, Inc., 257-269 (1992).

[15] Mirbel, I., Ralyté, J.: Situational method engineering: combining assembly-based and roadmap-driven approaches. Requirements Engineering V.11, Nº 1, (2006)

[16] MOdeling Software Kitt (MOSKitt), http://www.moskitt.org

[17] Niknafs, A., Ramsin, R.: Computer-Aided Method Engineering: An Analysis of Existing Environments. CAiSE, 525-540 (2008).

[18] Prakash, N.: Towards a Formal Definition of Methods. Requirements Engineering. 1: Vol. 2. - pp. 23-50 (1997)

[19] Ralyté, J., Rolland, C.: An Assembly Process Model for Method Engineering. CAiSe. - pp. 267-283 (2001)

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

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

[22] Si-Said, S., Rolland, C., Grosz, G.: MENTOR: A Computer Aided Requirements Engineering Environment CAiSE, 22-43 (1996)

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

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