Method Engineering

Mario Cervera Úbeda

This report presents a model-driven approach to support the design and implementation of software development methods and also describes how it has been implemented within the MOSKitt platform. Furthermore, this report presents an extension of the proposal to deal with method variability. This extension is based on the notion of method family.

1. A Model-Driven Approach for the Design and Implementation of Software Development Methods

1.1. Background

Method Engineering (ME) is the engineering discipline that covers the design, construction and adaptation of methods, techniques and tools for the development of information systems (Brinkkemper, 1996). The first research works developed in ME date back to the early nineties where Kumar and Welke established the basis of this area (Kumar & Welke, 1992). Thereafter, many research efforts have attempted to provide solutions to the challenges that ME entails. Some of the most relevant contributions are those by Brinkkemper, Prakash, Ralyté and Karlsson (Brinkkemper, 1996; Brinkkemper et al., 1999; Prakash, 1997; Prakash, 1999; Ralyté & Rolland, 2001; Karlsson & Agerfalk, 2004; Karlsson & Agerfalk, 2009). These works are based on a modular view of methods whereby methods are built by assembling different kinds of methods modules, namely method fragments, method blocks, method chunks and method components respectively. These proposals (among others) have contributed to establish a solid and wide theoretical basis in the ME field. However, industry uptake of ME approaches is being currently slow, as acknowledged in a recent state-of-the-art review (Henderson-Sellers & Ralyté, 2010). The main cause of this reality is the lack of adequate Computer-Aided Method Engineering (CAME) environments that allow putting ME theory into practice. Surveys such as the one presented in (Niknafs & Ramsin, 2008) put in evidence this reality and demonstrate that most of the software support is provided as prototype tools that do not cover the whole ME lifecycle. Examples of such prototypes are: MERU (Gupta & Prakash, 2001), Decamerone (Harmsen, 1997), Method Editor (Saeki, 2003) and MENTOR (Si-Said et al., 1996). Up until now, MetaEdit+ (Kelly et al., 1996) is the only tool that has been commercialized.

In this work we aim to overcome the limitations of current CAME technology by applying Model-Driven Development (MDD) techniques. Thus, we present a methodological framework and a supporting CAME environment that provide complete support to the design and implementation of methods in the context of MDD. The CAME environment has been implemented as an extension of the MOSKitt platform, since its plugin-based architecture and its integrated modelling tools turn it into a suitable software platform to support the proposal.

1.2. The Proposal

The ME proposal that is presented in this work is situated within the context of MDD and aims at providing support to the design and implementation of software development methods. According to MDD principles, the design of methods is performed by specifying methods as models (*Method Design* phase) and the implementation of these methods (i.e., the generation of the supporting CASE environments) is performed by means of model transformations (*Method Implementation* phase). These two phases are illustrated in figure 1 and summarized below. Further details are provided in (Cervera et al., 2010; Cervera et al., 2012).

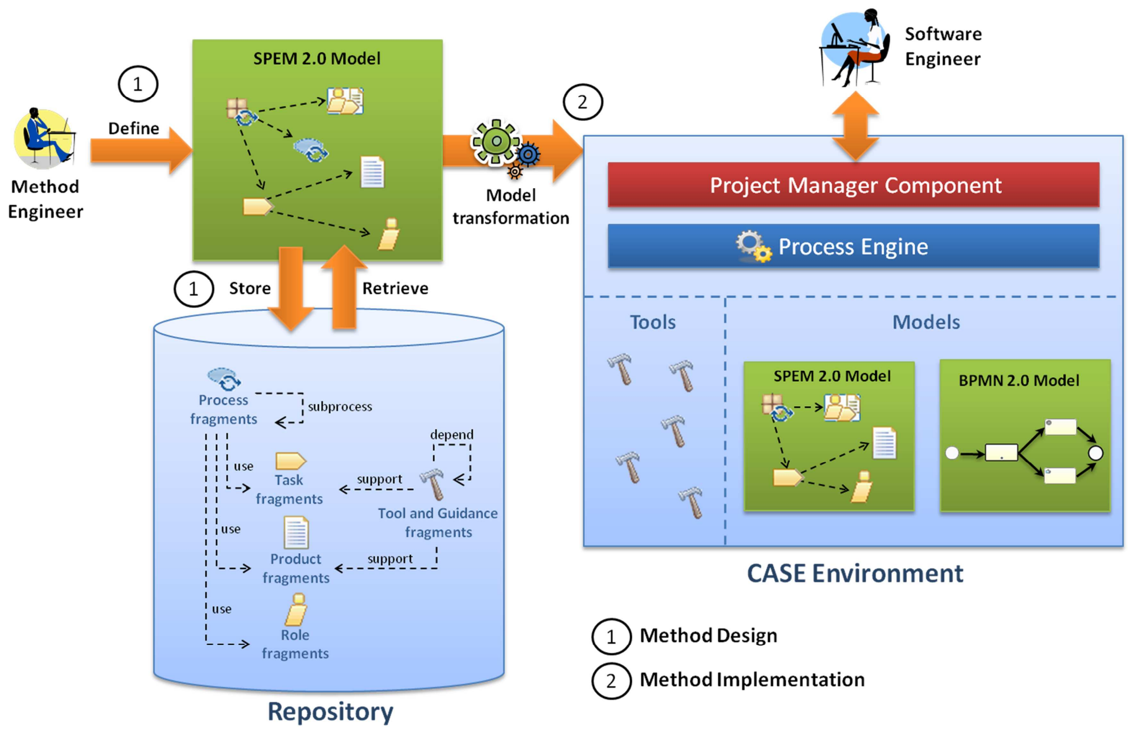


Figure 1. Proposal overview

Method Design

During the method design phase, the method engineer defines the method as a model by means of the SPEM 2.0 standard. The construction of this model can be performed from scratch or reusing method fragments that are retrieved from a repository. In addition, during the construction of this model the method engineer can store in the repository pieces of this model that will be available as method fragments for the construction of other method models.

In order to provide a more in-depth view on how method fragments are managed in our proposal, we present below a taxonomy that classifies the various types of fragments that are defined in our work and the relationships between them. Furthermore, the method fragments that represent SPEM 2.0 elements are associated with the SPEM 2.0 classes that denote the type of elements they contain.

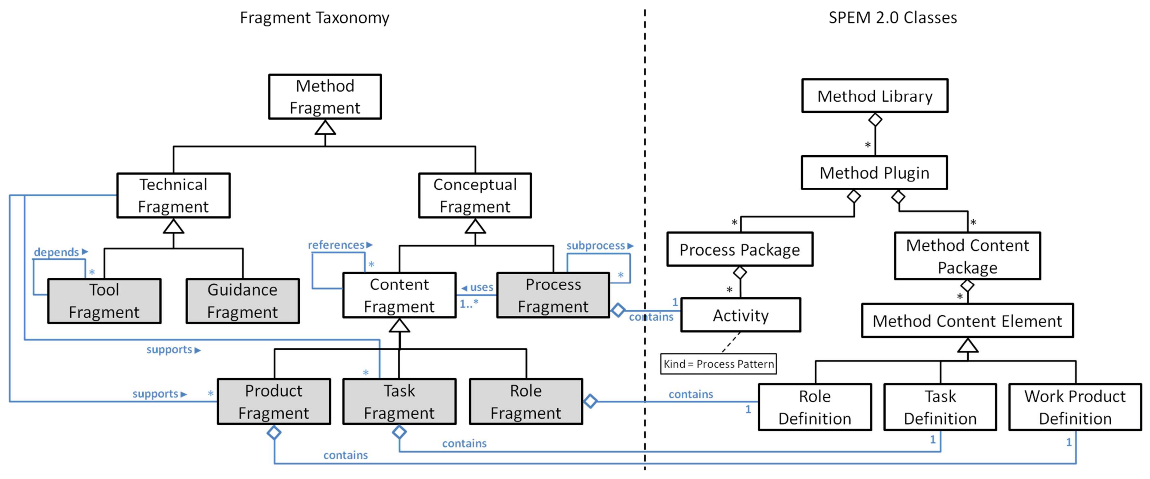


Figure 2. Fragment taxonomy and SPEM 2.0 classes

Figure 2 (left) shows the fragment taxonomy. As the figure shows, a method fragment can be either a conceptual fragment or a technical fragment. Conceptual fragments contain definitions of parts of methods such as roles, work products, tasks or processes. Technical fragments contain tools supporting the creation of method products. Conceptual fragments can be process fragments or content fragments. Process fragments contain reusable process patterns. Content fragments contain definitions of SPEM 2.0 method content, that is, tasks, work products and roles, which are stored respectively in task fragments, product fragments, and role fragments. On the other hand, technical fragments can be tool fragments or guidance fragments. Tool fragments contain software tools (e.g., editors or model transformations) that enable the creation of products. Guidance fragments contain guidelines about the performance of tasks. Finally, this taxonomy also illustrates relationships between method fragments. The “uses” relationship represents that a process pattern stored in a process fragment can make use of one or more elements contained in content fragments (tasks, products, etc.). Moreover, the “subprocess” relationship represents that a process pattern can be applied to build larger process patterns and the “references” relationship represents that a method content element may reference other method content elements (e.g., a task definition referencing its output work products definitions). The “depends” relationship represents that the tool contained in a tool fragment may depend on tools of other tool fragments for its correct operation. Last but not least, the “supports” relationship represents that products and tasks are associated with the technical fragments that will support them during the method enactment.

Figure 2 (right) shows a simplified view of the SPEM 2.0 meta-model. Some of the classes of this meta-model have been associated with the conceptual fragments of our taxonomy. These associations illustrate a containment relationship. For instance, process fragments are associated with the SPEM 2.0 class “Activity”. Thus, we are representing that process fragments contain a SPEM 2.0 model that includes one instance of this class. Note that in this case the “Activity” class is constrained. Specifically, process fragments must contain one instance of the class “Activity” that represents a SPEM 2.0 process pattern (also known as capability pattern).

In our proposal, method fragments are stored in the repository as reusable assets based on the RAS standard (Object Management Group, 2005). In order to allow method engineers to search and retrieve method fragments from the repository, the fragments are defined by a set of properties. These properties are stored in the manifest file of the RAS asset that embodies the fragment. The main properties characterizing method fragments are: "situation", "type", "origin", and "objective". Situation describes the context in which the fragment can be reused. Type describes the nature of the content of the fragment. For conceptual fragments, the valid types are "task", "role", "product" and "process". For technical fragments, type describes the kind of tool or guide included in the fragment, e.g., "editor", "model transformation", "cheat sheet", etc. Origin and objective describe respectively the method from which the fragment was created and the goal the fragment helps achieve.

Method Implementation

In this phase, a model transformation (Cervera et al., 2010) generates a CASE environment that provides support to the method that has been specified during the method design. As figure 3 shows, this CASE environment is composed of a dynamic part and a static part.

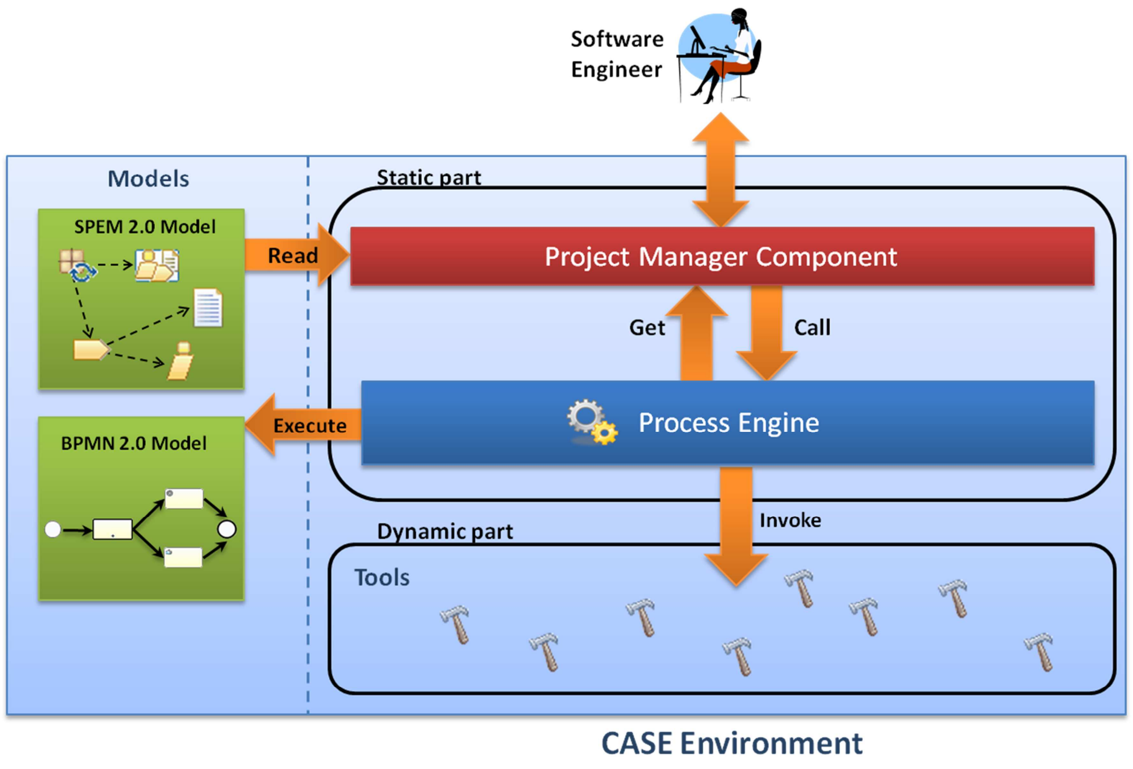


Figure 3. Generated CASE environment

The dynamic part is composed of those elements that are obtained from the method model and are, therefore, method-dependent. As figure 3 shows, the dynamic elements are the tools that provide software support to the method (i.e., graphical editors, model transformations, etc.). These tools are specified within the method model as technical fragments. Therefore, the model transformation must take these tools from the repository (and their dependencies) and integrate them in the generated CASE environment. In the context of MOSKitt, the tools contained in the technical fragments are implemented as Eclipse plugins and therefore their integration into the final CASE environment can be automatically performed since this tool is also based on Eclipse. Unfortunately, the integration of tools developed outside of the context of Eclipse cannot be guaranteed.

On the other hand, the static part is composed of those components that are always installed in the CASE tool irrespective of the method that has been specified. As figure 3 shows, these elements are the Project Manager Component (PMC) and the Process Engine. The PMC provides a GUI for the CASE environment and aims at assisting software engineers during the method enactment (i.e., during the course of the development projects). To achieve this goal, the PMC makes use of the SPEM 2.0 method model at runtime and also invokes the Process Engine, which is in charge of executing the method process part. For this purpose, the method process part must be represented in terms of an executable language, e.g. BPMN 2.0. Thus, in our proposal a BPMN 2.0 model is obtained from the SPEM 2.0 method model during the CASE tool generation process. Further details about the PMC (and the method implementation phase in general) are provided in (Cervera et al., 2012).

1.3. MOSKitt4ME : Implementation of the Proposal in MOSKitt

The proposal presented in the previous section has been implemented as an extension of the MOSKitt tool. This extension is called MOSKitt4ME and its architecture is depicted in figure 4.

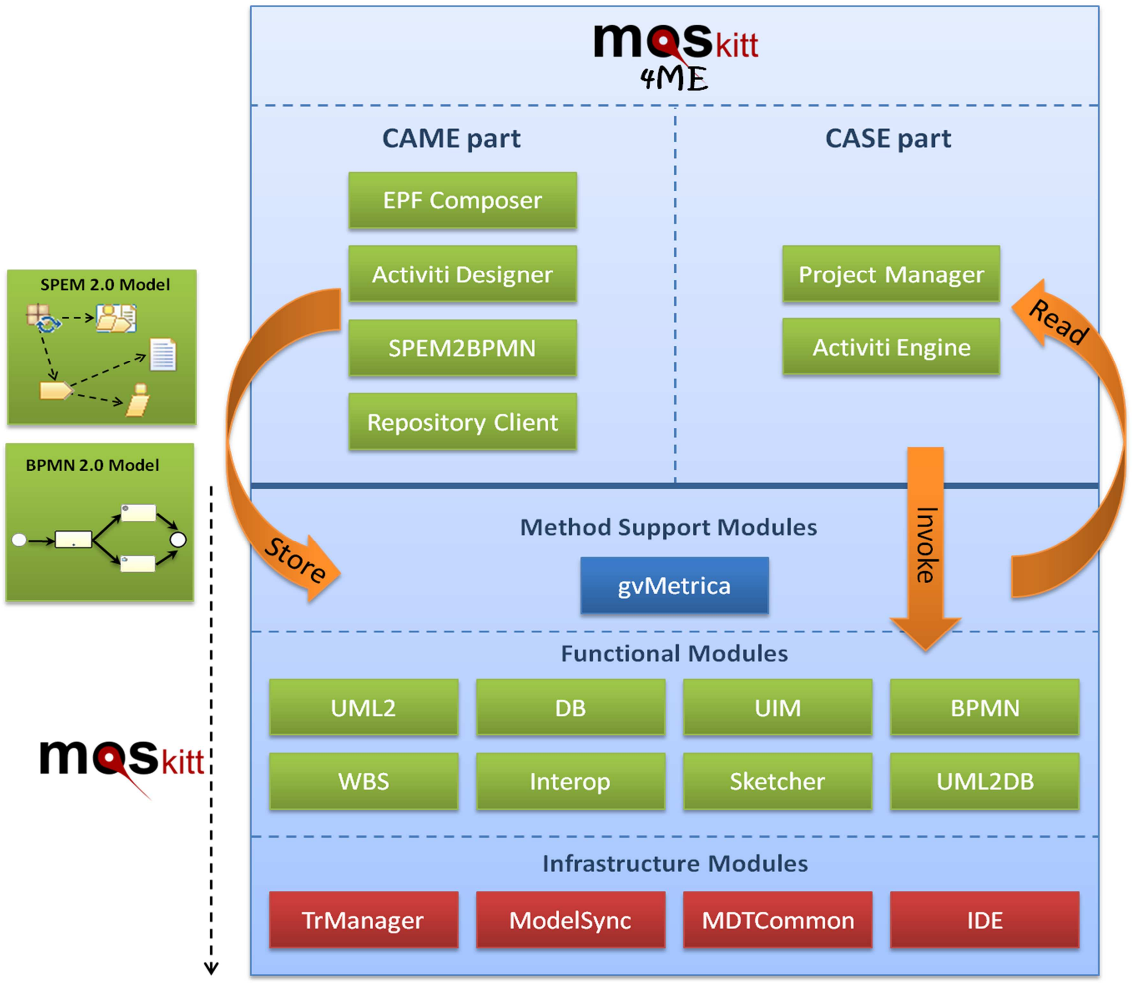


Figure 4. MOSKitt4ME architecture

Specifically, MOSKitt4ME is built upon MOSKitt modules, which are divided into three groups:

- **Infrastructure modules**: these modules provide facilities for processing, manipulating and managing MOSKitt artifacts such as models or model transformations. For instance, the "TrManager" module provides mechanisms to declare and invoke model transformations and the "ModelSync" module implements the infrastructure needed to define the synchronization between models that have been transformed and traced.
- **Functional modules**: these modules provide specific tools that allow creating different kind of resources such as models, diagrams or text files. For instance, the "UML2" module implements a graphical editor that allows creating UML 2.0 diagrams.
- **Method support modules**: these modules provide support to the gvMetrica method, in particular for defining and running its development processes. These processes are defined by means of the Dashboard tool and executed by means of the Dashboard Interpreter.

In this context, MOSKitt4ME modules aim to improve the method support provided in MOSKitt by the Dashboard and Dashboard Interpreter tools. The Dashboard tool provides a graphical editor for defining software development methods. However, this tool is very limited in terms of support for modeling the elements of gvMetrica. On the other hand, the Dashboard Interpreter provides an execution environment for enacting method instances. However, this tool has not been successful in terms of user satisfaction due to its low usability.

In order to solve these problems, MOSKitt4ME provides a ME environment that supports the definition of methods by means of the SPEM 2.0 standard, and also the enactment of method instances. For this purpose, MOSKitt4ME defines six modules. The first four constitute the **CAME part** of MOSKitt4ME, since they are focused on method design, and the last two modules constitute the **CASE part,** since they aim at assisting software engineers during the method enactment. These modules are the following:

- **EPF Composer**: an SPEM 2.0 editor that is provided as part of the Eclipse Process Framework (EPF) Project and has been integrated in MOSKitt to allow method engineers to create method models based on the SPEM 2.0 standard.
- **Activiti Designer**: a BPMN 2.0 graphical editor that is provided as part of the Activiti Project and has been integrated in MOSKitt to allow method engineers to define processes based on the BPMN 2.0 standard.
- **SPEM2BPMN**: a Model-To-Model (M2M) transformation that obtains from a SPEM 2.0 model an executable representation of the process part in terms of BPMN 2.0.
- **Repository client**: an Eclipse view that allows the method engineer to connect with the repository to store or retrieve method fragments. This component is not yet implemented in the current version of MOSKitt4ME.
- **Project manager (PMC)**: provides a set of Eclipse views that assist software engineers during the method enactment (Cervera et al., 2012).
- **Activiti Engine**: a process engine that is provided as part of the Activiti Project and has been integrated in MOSKitt to enable the execution of BPMN 2.0 processes defined by means of the Activiti Designer.

These modules work as follows. The method engineer defines the method (in this case gvMetrica) by means of the EPF Composer. The repository client can be used to search for method fragments and integrate them in the method model under construction (and also to store new method fragments in the repository). Once the method is finished, the SPEM2BPMN transformation allows the method engineer to obtain an executable representation of the method process part in terms of BPMN 2.0. This process definition can be manually enhanced by means of the Activiti Designer. This is often needed since BPMN 2.0 provides more expressiveness than SPEM 2.0 with respect to process elicitation. Then, the SPEM 2.0 and BPMN 2.0 models are stored in the gvMetrica module. The Project Manager and Activiti Engine read these models from the gvMetrica module to perform the method enactment (i.e., to assist software engineers during the course of the software development projects).

It is important to highlight that in MOSKitt4ME the transformation that generates CASE environments (see subsection 1.2) is not implemented since MOSKitt4ME is intended for supporting gvMetrica and all the required tools are already installed in MOSKitt (the functional modules). Thereby, only the static part of the CASE tool (the PMC and the process engine) is needed.

2. Incorporating Variability

2.1. Background

Theoretically, software development methods define complete and integrated approaches for building software systems. However, practice has proven that a particular method is only suitable for a very specific project type whereas software companies normally deal with a high diversity of software projects. Sources of these diversities may be differences among project characteristics such as domains of the systems to be developed (e.g., web application, mobile application, etc.), development lifecycles (e.g., waterfall, spiral, etc.), budget, project size, project duration, etc. Due to these diversities, software companies are forced to adapt their methods to the particular situations where the methods are to be applied. Thus, software companies make use of a multitude of method variants, whereby each of these variants is valid for a specific context or project type. In this scenario, which solutions do software companies use to adequately handle method variability?

A common approach to manage method variability is keeping all method variants in separate method models. This approach is called "multi-model approach" (Hallerbach et al., 2010). Nonetheless, this solution is only feasible if few variants exist or they differ to a large degree from each other. Otherwise, a multi-model approach will bring high maintenance costs as well as redundancies and inconsistencies between method models. A better approach is to combine principles of Method Engineering and Software Product Line Engineering (SPLE), which leads to the notion of method family. Method families have been recently proposed in (Asadi et al., 2011; Kornyshova et al., 2011) and are defined by analogy to the notion of product family (Gurp, 2001) as a set of related methods that share common parts while others are variable. These common and variable parts can be organized in a variability model from which specific methods can be configured according to project characteristics. This approach allows avoiding the aforementioned drawbacks since the common and variable parts are specified separately and are not replicated.

Since we consider that method families represent an adequate solution to handle method variability, we extend our ME approach to incorporate support for method families. With our solution we intend to go a step beyond related work, providing the following contributions:

1. A methodological approach that not only supports the specification of method families but also the generation of the supporting CASE environments. Thereby, method and software engineers are able to properly handle method variability both at method design and method implementation levels. Current approaches such as (Asadi et al., 2011) and (Kornyshova et al., 2011) only cover method family specification.
2. A methodological approach that provides support to the process part of method families. Current approaches define method families that allow the method engineer to configure specific methods by assembling common and variable parts. However, they do not support a complete specification of the control-flow that governs the execution order of these parts (i.e., their orchestration).
3. A CAME environment that provides complete software support to our methodological approach. To the best of our knowledge, it has not yet been developed a CAME environment that provides support to method families.

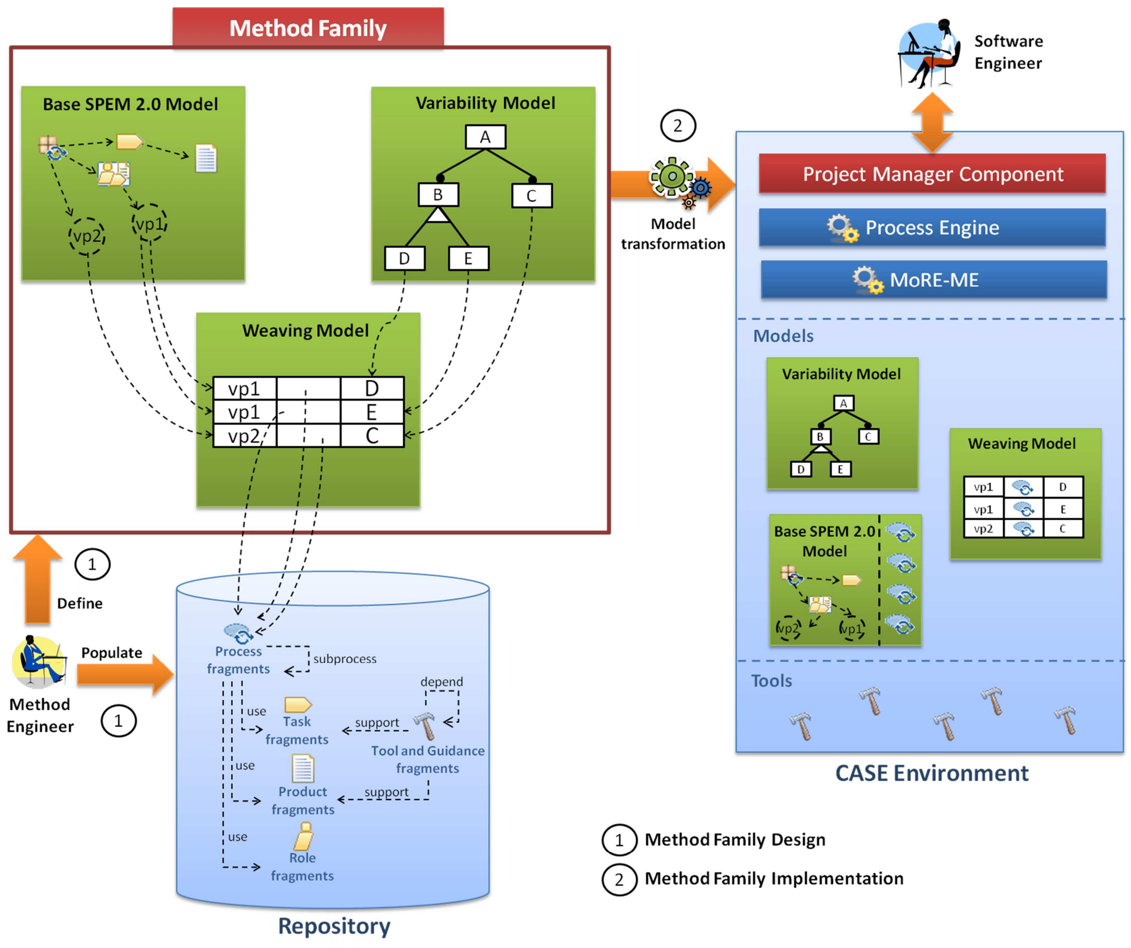


Figure 5. Proposal overview (refined to support variability)

2.2. Proposal Extension

We extend our ME proposal to support the definition of method families and the generation of the supporting CASE environments. Thereby, we now divide our approach in two phases: *Method Family Design* and *Method Family Implementation*. These two phases are illustrated in figure 5 and detailed below.

Method Family Design

During this phase the method engineer defines the method family. In our approach, this is performed in two steps.

Step 1) Population of the repository

In this step, the method engineer populates the repository with the reusable components (in this case method fragments) that are required for a complete specification of the method family. In other words, the method engineer builds (if they are not already available in the repository) the conceptual fragments that represent the common and variable parts of the family and also develops, together with software engineers, the technical fragments that provide software support to the conceptual fragments. A key benefit of developing components that are reusable is that the method family can be easily used to build new methods at low cost and high quality.

Step 2) Definition of the method family

In this step, the method engineer builds the set of models that represent the method family. Firstly, the method engineer defines either from scratch or reusing method fragments a base method model by means of SPEM 2.0. The **base method model** contains the commonalities that share all method variants and also the specification of those parts that are subject to variation (i.e., variation points). A variation point can be defined as a precise position within a method model that admits different possibilities according to the current context or situation (Ayora, 2011). In SPEM 2.0, variation points can be specified by means of empty activities (Martínez-Ruíz et al., 2009). When a specific method is later configured, process patterns will be applied in these activities.

In parallel to (or after) the definition of the base method model, the method engineer must define the variability model. In our approach, the **variability model** is defined as a feature model (Lee et al., 2002). In SPLE, feature modeling is the activity of identifying externally visible characteristics of software products (i.e, features) and organizing them into a model called a feature model. In a feature model, common features among different products are modeled as mandatory features, while variable features are modeled as optional or alternative. By analogy to SPLE, in the context of method families a feature model must gather all features of methods that allow the method engineer to configure different method variants. Since we consider that there is a strong relation between project characteristics and method variability (that is to say, changes in the project characteristics are the triggers of method variation), we suggest that features in the feature model represent project characteristics. Specifically, project characteristics are represented as interior nodes in the feature model and the project characteristics possible values are represented as variants of these interior nodes. These variants in the feature model must be linked with the variation points of the base method model that are subject to variation according to the particular project characteristic values that the variants represent. Furthermore, each variation point (and variant) must be associated with one or more process fragments that contain the process patterns that can be applied in the variation point[1] (if the specific variant is active). For the association of process fragments, variants and variation points, we suggest the use of a weaving model. The **weaving model** allows associating all these elements in a loosely coupled manner, that is, without adding any further information to the related models.

As an example, consider a software company that develops systems for two different domains, e.g., mobile applications and embedded systems. In this context, method engineers determine that “domain” is a significant characteristic of the projects that are carried out in the company and therefore it is included in the variability model as a feature (interior node). This feature will contain two variants, namely “mobile application” and “embedded system”. Now, let us consider that there is only one variability point, called “system design”, in the base method of the company that is affected by this project characteristic. This variability point is modeled as an empty activity where a different process pattern will be applied depending on the value of the “domain” feature. In other words, depending on whether a project is for developing a mobile application or an embedded system, the “system design” activity will be performed differently. In addition, let us consider that the repository contains two process fragments that define (as process patterns) two different ways of performing the system design. These fragments are called respectively “design for mobile applications” and “design for embedded systems”. According to this scenario, the weaving model will contain the following associations between elements:

1. “system design” - “design for mobile applications” - “mobile application”
2. “system design” - “design for embedded systems” - “embedded system”

For instance, the first association is read as follows. In the variability point “system design”, the process pattern “design for mobile applications” will be applied when the feature “mobile application” is active.

Method Family Implementation

In this phase, a CASE environment supporting the method family specified in the previous phase is generated by means of a model transformation. Similarly to the tool illustrated in figure 3, the infrastructure of this CASE environment is composed of the Project Manager Component, the Process Engine and the tools supporting the tasks and products of the method family (the technical fragments). In addition, the CASE environment also includes the three models that define the method family so that they can be used at runtime. These three models are included “as is” and then the process fragments referenced in the weaving model (and the other fragments associated to these process fragments according to the relationships illustrated in figure 2) are integrated in the base method model. These elements (process patterns, tasks, roles, etc.) are required in order to enable the automatic generation of specific

[1] Note that it is sufficient to link features with process fragments because all other type of fragments can be reached from them thanks to the relationships defined in figure 2.

method models from the base method model (see below). Finally, unlike the CASE environment depicted in figure 3, CASE environments now include a model-based reconfiguration engine, called MoRE-ME, which is an extension of MoRE (Cetina et al., 2009) for ME. MoRE-ME allows method and software engineers to deal with method variability, since it enables the dynamic reconfiguration of method variants when changes in the project characteristics arise. Thereby, we provide support to the dynamic nature of projects, an issue that is not yet properly addressed in the ME field (Rolland, 2009). Moreover, projects based on different methods can coexist within the same CASE environment. This was not supported in the previous version of our proposal.

Figure 6 illustrates the architecture of the CASE environments that are generated in the method family implementation phase. This architecture is an extension of the architecture shown in figure 3. Within these CASE environments, software engineers create new development projects, specify their characteristics and dynamically configure specific methods suited to the projects where they are to be applied. Once a specific method is configured for a particular project, the CASE environment assists software engineers during the method enactment. All the steps involved from the creation of a project to the enactment of its specific development method are indicated in figure 6 and detailed below.

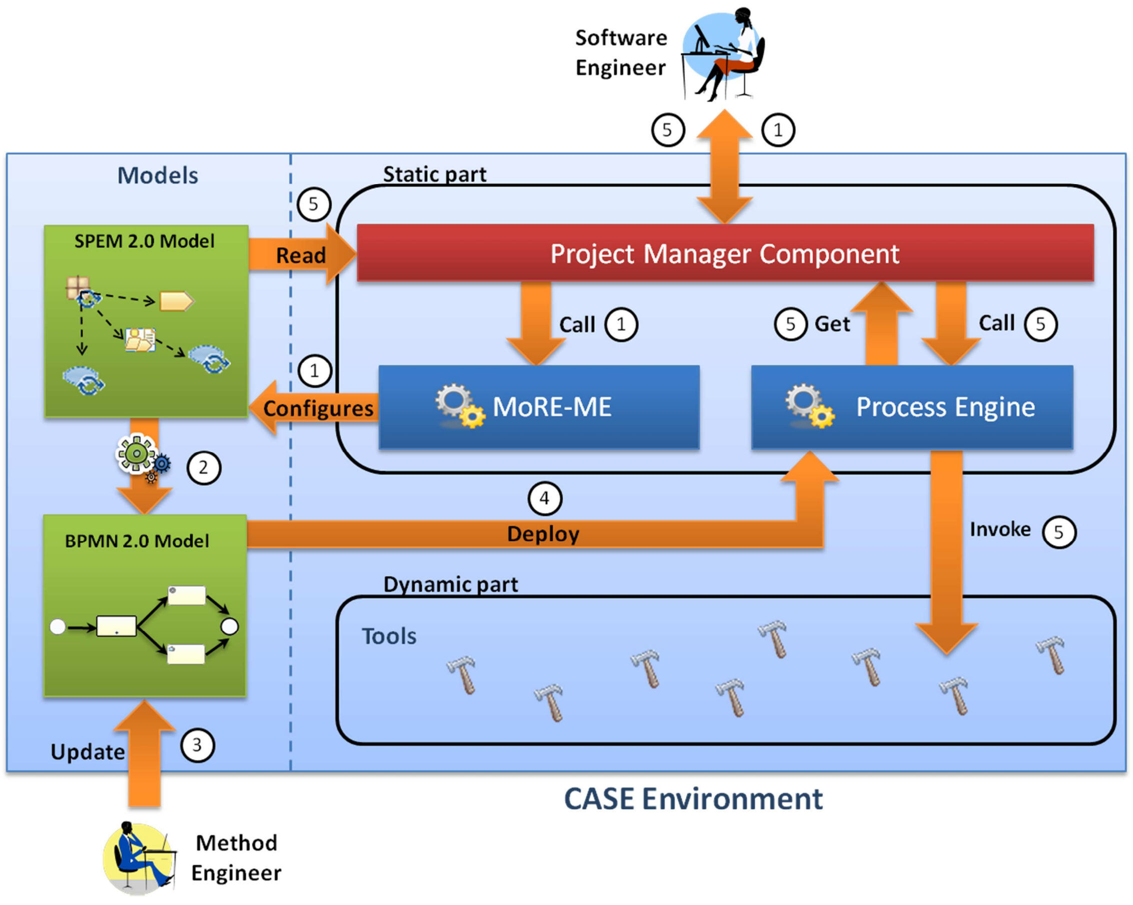


Figure 6. Generated CASE environment (refined to support variability)

Step 1) Specification of a software project

This step involves the creation of a new project by means of the PMC or the modification of the characteristics of an existing project. In both cases, a specific method that governs the execution of the project must be obtained. In the first case, a new method will be obtained from the base method model. In the second case, the specific method model associated to the project will be reconfigured. For this purpose, in this step the software engineer must specify the project characteristics. These characteristics define the type of project that is to be performed and, therefore, they allow obtaining the appropriate method variant from the method family. The configuration of a specific method variant according to the project characteristics is automatically performed by MoRE-ME. This is illustrated in figure 7.

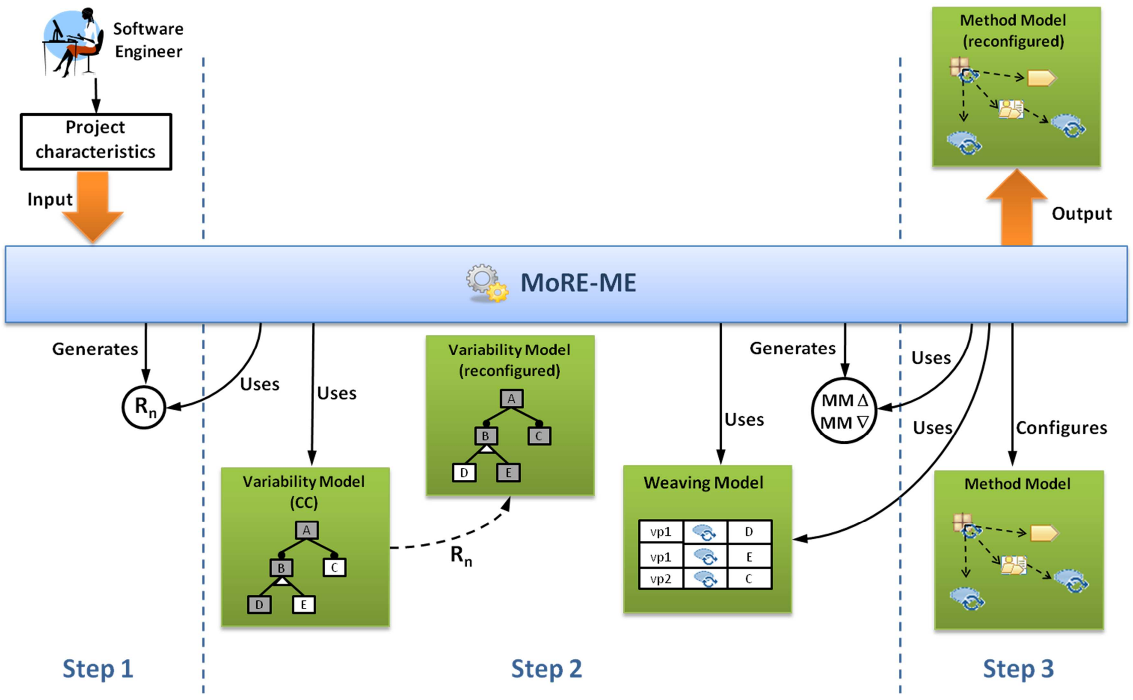


Figure 7. Configuration of methods by MoRE-ME

As figure 7 shows, in a first step MoRE-ME translates the project characteristics into a resolution. Resolutions are used by MoRE-ME to (de)activate features of the feature model. Specifically, a resolution is a list of pairs (F, S) in which F indicates a feature and S the feature's state (Cetina et al., 2009). More formally, let FM be a feature model, then:

$$R = \{(F, S)\} \mid F \in FM \wedge S \in \{Active, Inactive\}$$

Thereby, when the software engineer specifies the characteristics of the project, MoRE-ME generates a resolution called R_n (n indicates that the project characteristics have changed for the nth time, thus being R_0 the initial resolution for new projects). This resolution is composed of those features (and variants) that represent the project characteristics (and their values) specified by the method engineer. For instance, a possible resolution R_0 for the example of the previous subsection is:

$$R_0 = \{(Domain, Active), (Mobile\ application, Active), (Embedded\ System, Inactive)\}$$

This resolution is used by MoRE-ME to update the current configuration of the feature model (step 2). The current configuration (CC) indicates the set of all active features of the feature model (Cetina et al., 2009). More formally, let FM be a feature model, then:

$$CC = \{F\} \mid F \in FM \wedge F.state = Active$$

Also in step 2, MoRE-ME makes use of the CC, the resolution R_n and the weaving model to calculate the increments and decrements to be applied on the method model[2]. More formally, the increments and decrements to be applied on the method model (MM) are defined as follows. Let PP be an operation that returns the process patterns associated to a feature of the feature model, then:

$$MM\Delta = PP\ (F \mid (F, S) \in R_n \wedge S = Active \wedge F \notin CC)$$

$$MM\nabla = PP\ (F \mid (F, S) \in R_n \wedge S = Inactive \wedge F \in CC)$$

$MM\Delta$ is the set of all process patterns to be applied on the method model. Specifically, this set is composed of those patterns that are associated to the features of R_n that are set as active and are not included in the current configuration. On the other hand, $MM\nabla$ is the set of all process patterns to be removed from the method model. Specifically, this set is composed of those patterns that are associated to the features of R_n that are set as inactive and are included in the current configuration.

Finally, the last step involves the application and removal of the process patterns contained respectively in $MM\Delta$ and $MM\nabla$. In case of the creation of a new project ($n = 0$), the specific method model of the project will be obtained just by applying $MM\Delta$ on the base method model. However, in case of the modification of project characteristics ($n \geq 1$), the specific method model already exists. Therefore, this model must be reconfigured by removing the process patterns contained in $MM\nabla$ and applying the process patterns contained in $MM\Delta$.

It is very important to highlight that when $n \geq 1$ only the process patterns involved in the part of the method that has not been executed will be added or removed. In other words, the part of the method that has already been executed remains unmodified. This is important in order to enable the automatic update of the process instance associated to the project.

Step 2) Generation of the executable process

Before deploying the specific method model obtained in step 1 into the process engine, an executable representation of the method process part must be obtained. Specifically, we suggest mapping the method process part into BPMN 2.0 by means of a M2M transformation. Further details about this step are provided in (Cervera et al., 2012).

Step 3) Manual update of the executable process

The executable representation of the process in terms of BPMN 2.0 can be manually modified to represent more complex workflows. This is usually needed since BPMN 2.0 provides more

[2] This method model will be the base method model for R_0 and a specific method model for R_n where $n \geq 1$.

expressiveness than SPEM 2.0 with respect to process elicitation. Further details about this step are provided in (Cervera et al., 2012).

Step 4) Process deployment

Once the process specification is finished, it must be deployed in the process engine so that the PMC can execute process instances. If n ≥ 1 (see figure 7), when the process instance is created after deployment, it must be updated to reflect the previous state of the project. This involves marking as executed the tasks that were executed in the previous process instance.

Step 5) Method use

Once the process has been deployed and the process instance has been started, both the process engine and the SPEM 2.0 model are used by the PMC to assist software engineers during the project lifecycle (Cervera et al., 2012). It is important to highlight that during the course of the projects, the project characteristics that were selected during their creation may change. In this case, the five steps illustrated in figure 6 must be performed again in order to reconfigure the specific method model associated to the project (i.e., to adapt the method to the new characteristics of the project).

2.3. Variability in MOSKitt4ME

Figure 8 shows the new architecture of MOSKitt4ME for supporting variability. As the figure shows, the **CAME part** is now composed of the MOSKitt Feature Modeler, the EPF Composer, the ATLAS Model Weaver (AMW) and the Repository Client. These components allow the method engineer to define the three models that compose the gvMetrica method family, that is, the feature model, the base SPEM 2.0 model and the weaving model. The feature model defines the characteristics of the different types of projects that can be performed in the context of the CIT. The base SPEM 2.0 model defines a method that embodies the common part of all method variants of the family. This model also defines the variation points (as empty activities) and the process patterns that can be applied in the variation points. Finally, the weaving model contains the associations between project characteristics (features), variation points and process patterns.

On the other hand, the **CASE part** is now composed of the PMC, the Activiti Engine, the Activiti Designer, the SPEM2BPMN transformation and MoRE-ME. These tools allow software and method engineers to perform the steps described in the previous subsection.

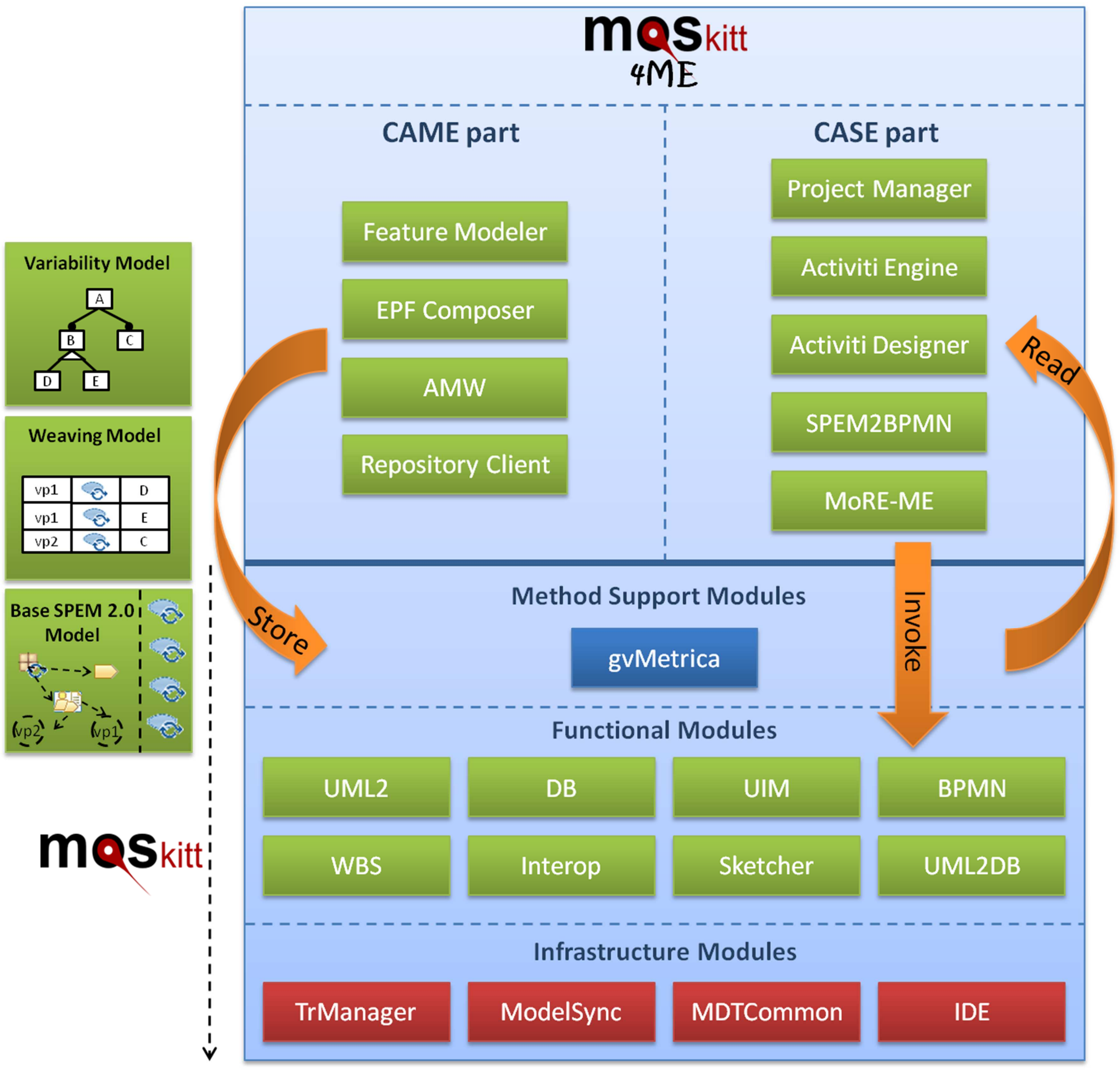
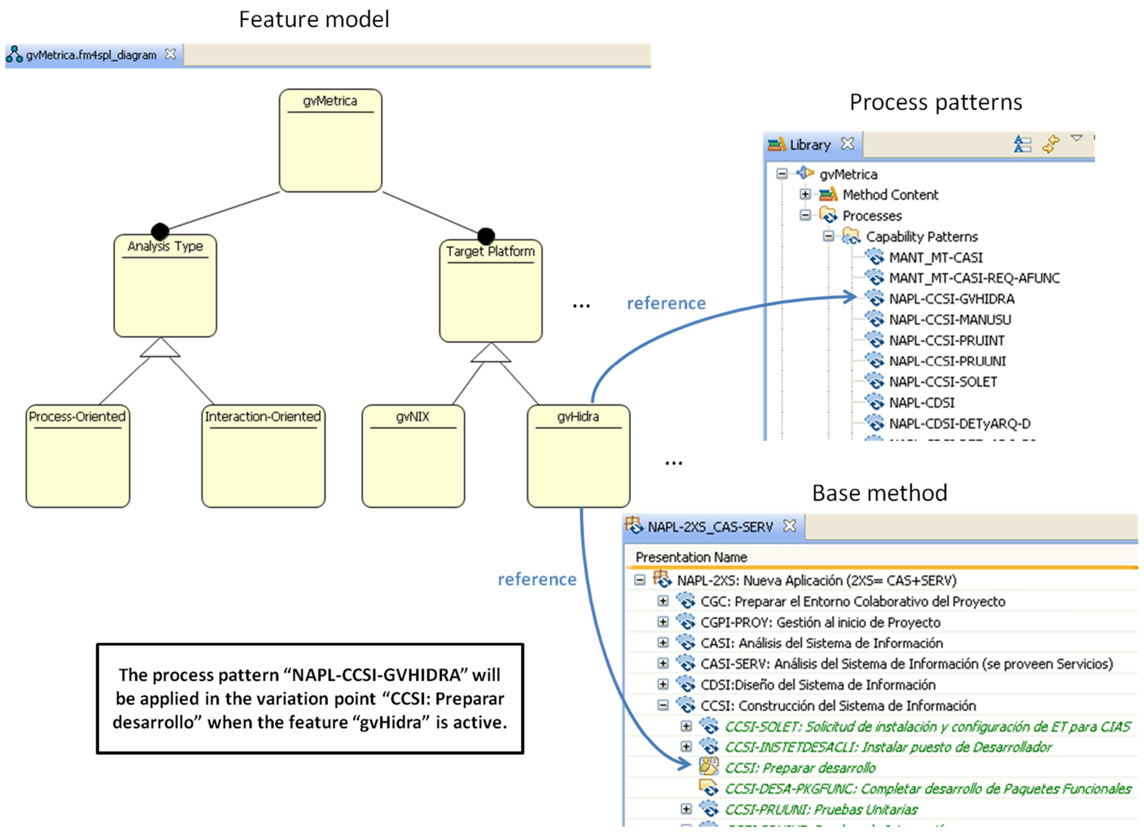


Figure 8. MOSKitt4ME architecture (refined to support variability)

An example

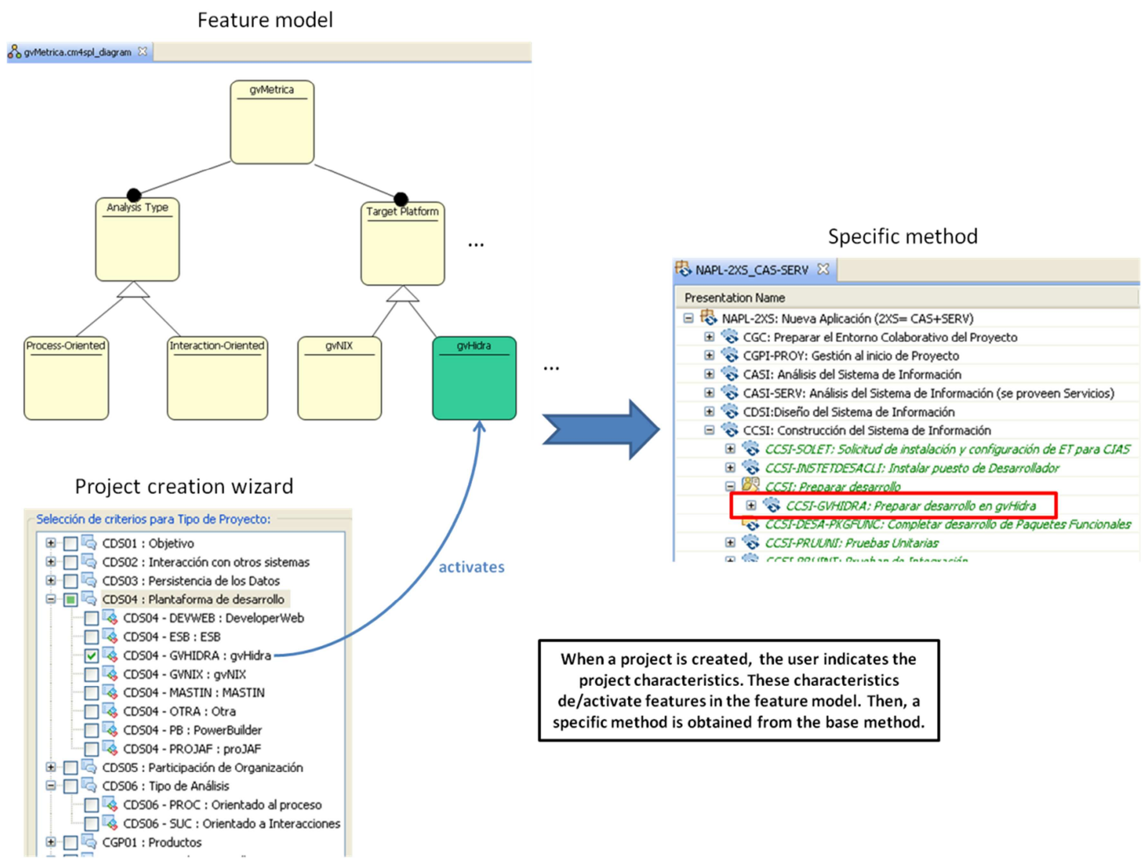
This subsection presents a brief but illustrative example of some of the ideas put forward in this report. In figure 9 one can see an excerpt of a feature model created by means of the Feature Modeler. This feature model contains some of the project characteristics that are considered relevant within the CIT. For instance, since different method variants are used when the target platform of the software varies, the feature model contains a feature called "Target Platform". This feature contains two alternative variants "gvNIX" and "gvHidra". In addition, figure 9 also shows two excerpts of the base SPEM 2.0 model. The upper-right corner of the figure shows some process patterns. Specifically, the process pattern "NALP-CCSI-GVHIDRA" has been associated (by means of a weaving model, not shown in the example) with the "gvHidra" feature. Thus, the method engineer is indicating that this process pattern must be applied on the base method model when the software engineer creates a new project for developing an application whose target platform is gvHidra. The lower-right corner shows an excerpt of the base method that contains a variation point called "CCSI: Preparar desarrollo". The feature "gvHidra" has been associated with this variation point. Thus, the method engineer is indicating that the process pattern “NAPL-CCSI-GVHIDRA” will be applied in this variation point when the feature “gvHidra” is activated.


Feature model
gvMetrica.fm4spl_diagram
gvMetrica
Analysis Type
Target Platform
Process-Oriented
Interaction-Oriented
gvNIX
gvHidra
...
reference
Process patterns
Library
gvMetrica
Method Content
Processes
Capability Patterns
MANT_MT-CASI
MANT_MT-CASI-REQ-AFUNC
NAPL-CCSI-GVHIDRA
NAPL-CCSI-MANUSU
NAPL-CCSI-PRUINT
NAPL-CCSI-PRUUNI
NAPL-CCSI-SOLET
NAPL-CDSI
NAPL-CDSI-DETyARQ-D
...
Base method
NAPL-2XS_CAS-SERV
Presentation Name
NAPL-2XS: Nueva Aplicación (2XS= CAS+SERV)
CGC: Preparar el Entorno Colaborativo del Proyecto
CGPI-PROY: Gestión al inicio de Proyecto
CASI: Análisis del Sistema de Información
CASI-SERV: Análisis del Sistema de Información (se proveen Servicios)
CDSI:Diseño del Sistema de Información
CCSI: Construcción del Sistema de Información
CCSI-SOLET: Solicitud de instalación y configuración de ET para CIAS
CCSI-INSTETDESACLI: Instalar puesto de Desarrollador
CCSI: Preparar desarrollo
CCSI-DESA-PKGFUNC: Completar desarrollo de Paquetes Funcionales
CCSI-PRUUNI: Pruebas Unitarias
reference

The process pattern “NAPL-CCSI-GVHIDRA” will be applied in the variation point “CCSI: Preparar desarrollo” when the feature “gvHidra” is active.

Figure 9. Example of method family

The application of the process pattern is illustrated in figure 10. Specifically, the PMC provides a project creation wizard that allows software engineers to specify the project characteristics. In the example, the software engineer has specified that the project to be created is for developing an application based on gvHidra. When this project characteristic is selected, MoRE-ME activates the corresponding feature in the feature model. Then, it applies the process pattern in the base method to obtain the specific method model that will govern the execution of the project.


Feature model
gvMetrica.cm4spl_diagram
gvMetrica
Analysis Type
Target Platform
...
Process-Oriented
Interaction-Oriented
gvNIX
gvHidra
...
Project creation wizard
Selección de criterios para Tipo de Proyecto:
CDS01 : Objetivo
CDS02 : Interacción con otros sistemas
CDS03 : Persistencia de los Datos
CDS04 : Plantaforma de desarrollo
CDS04 - DEVWEB : DeveloperWeb
CDS04 - ESB : ESB
CDS04 - GVHIDRA : gvHidra
CDS04 - GVNIX : gvNIX
CDS04 - MASTIN : MASTIN
CDS04 - OTRA : Otra
CDS04 - PB : PowerBuilder
CDS04 - PROJAF : proJAF
CDS05 : Participación de Organización
CDS06 : Tipo de Análisis
CDS06 - PROC : Orientado al proceso
CDS06 - SUC : Orientado a Interacciones
CGP01 : Productos
activates
Specific method
NAPL-2XS_CAS-SERV
Presentation Name
NAPL-2XS: Nueva Aplicación (2XS= CAS+SERV)
CGC: Preparar el Entorno Colaborativo del Proyecto
CGPI-PROY: Gestión al inicio de Proyecto
CASI: Análisis del Sistema de Información
CASI-SERV: Análisis del Sistema de Información (se proveen Servicios)
CDSI:Diseño del Sistema de Información
CCSI: Construcción del Sistema de Información
CCSI-SOLET: Solicitud de instalación y configuración de ET para CIAS
CCSI-INSTETDESACLI: Instalar puesto de Desarrollador
CCSI: Preparar desarrollo
CCSI-GVHIDRA: Preparar desarrollo en gvHidra
CCSI-DESA-PKGFUNC: Completar desarrollo de Paquetes Funcionales
CCSI-PRUUNI: Pruebas Unitarias

When a project is created, the user indicates the project characteristics. These characteristics de/activate features in the feature model. Then, a specific method is obtained from the base method.

Figure 10. Application of a process pattern

References

Asadi, M., Mohabbati, B., Gasevic, D., Bagheri, E.: Developing Families of Method-Oriented Architecture. Engineering Methods in the Service-Oriented Context, Springer Berlin, Heidelberg, 351, 168-183 (2011)

Ayora, C.: Modelling and Managing Variability in Business Processes. Master’s Thesis. Universitat Politècnica de València (2011)

Brinkkemper, S.: Method Engineering: Engineering of Information Systems Development Methods and Tools. Information and Software Technology, 38, 275-280 (1996)

Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-modelling based assembly techniques for situational method engineering. Inf. Syst. 24, 209-228 (1999)

Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. Computer, IEEE Computer Society Press, 42, 37-43 (2009)

Cervera, M., Albert, M., Torres, V., Pelechano, V.: A methodological framework and software infrastructure for the construction of software production methods. In: Yang, Y., Münch, J., Schäfer, W. (eds.) ICSP 2010. LNCS, vol. 6195, pp. 112-125. Springer-Verlag, Berlin, Heidelberg (2010)

Cervera, M., Albert, M., Torres, V., Pelechano, V., Bonet, B., Cano, J.: A technological framework to support model driven method engineering. In Actas de los Talleres de las Jornadas de Ingeniera del Software y Bases de Datos, 47-56 (2010)

Cervera, M., Albert, M., Torres, V., Pelechano, V.: A Model-Driven Approach for the Design and Implementation of Software Development Methods. To be published in: International Journal of Information System Modeling and Design (2012)

Gupta, D., Prakash, N.: Engineering Methods from Method Requirements Specifications. Requirements Engineering, 6, 135-160 (2001)

Gurp, J. V., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, 45- 54 (2001)

Hallerbach, A., Bauer, T., Reichert, M.: Configuration and Management of Process Variants. International Handbook on Business Process Management I, Springer, 237-255 (2010)

Harmsen, A.F.: Situational Method Engineering. Ph.D. thesis, Univ. of Twente, Utrecht (1997)

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

Karlsson, F., Agerfalk, P.J.: Method configuration: adapting to situational characteristics while creating reusable assets. Information and Software Technology 46, 619-633 (2004)

Karlsson, F., Agerfalk, P.J.: Towards structured flexibility in information systems development: Devising a method for method configuration. J. Database Manag. 20, 51-75 (2009)

Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In Constantopoulos, P.; Mylopoulos, J. & Vassiliou, Y. (ed.). Advanced Information Systems Engineering, 1080, 1-21 (1996)

Kornyshova, E., Deneckère, R., Rolland, C.: Method Families Concept: Application to Decision-Making Methods. Enterprise, Business-Process and Information Systems Modeling, Springer Berlin Heidelberg, 81, 413-427 (2011)

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)

Lee, K., Kang, K. C., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, Springer-Verlag, 62-77 (2002)

Martínez-Ruíz, T., García, F., Piattini, M.: Modelado de Líneas de Procesos mediante SPEM v2.0. Conferencia iberoamericana de Software Engineering, 195-207, (2009)

Niknafs, A., Ramsin, R.: Computer-Aided Method Engineering: An Analysis of Existing Environments. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 525–540. Springer, Heidelberg (2008)

Object Management Group. (2007). Software & Systems Process Engineering Metamodel (v2.0). http://www.omg.org/spec/SPEM/2.0/

Object Management Group. (2005). Reusable Asset Specification (v2.2). http://www.omg.org/spec/RAS/

Prakash, N.: Towards a formal definition of methods. Requir. Eng. 2, 23-50 (1997)

Prakash, N.: On method statics and dynamics. Inf. Syst. 24, 613-637 (1999)

Ralyté, J., Rolland, C.: An approach for method reengineering. In: S.Kunii, H., Jajodia, S., Solvberg, A. (eds.) ER 2001, LNCS, vol. 2224, pp. 471-484. Springer Berlin / Heidelberg (2001)

Ralyté, J., Rolland, C.: An assembly process model for method engineering. In: Dittrich, K., Geppert, A., Norrie, M. (eds.) Advanced Information Systems Engineering, LNCS, vol. 2068, pp. 267-283. Springer Berlin / Heidelberg (2001)

Rolland, C.: Method engineering: towards methods as services. Software Process: Improvement and Practice 14(3), 143-164 (2009)

Saeki, M.: CAME: The First Step to Automated Method Engineering. In Workshop on Process Engineering for Object-Oriented and Component-Based Development (2003)

Si-Said, S.; Rolland, C., Grosz, G.: MENTOR: A Computer Aided Requirements Engineering environment. Advanced Information Systems Engineering, 1080, 22-43 (1996)