Mario Cervera Úbeda

Model Driven Method Engineering. A Supporting Infrastructure

Master's Thesis

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información - December 2010

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Supervisor

Vicente Pelechano Ferragud

Co-directors

Manuela Albert Albiol, Victoria Torres Bosch

To my family

Abstract

The Method Engineering discipline emerged two decades ago to face up to the challenge of defining software production methods, adapting them to fit particular project needs and building the supporting CASE tools. Over these twenty years many theoretical proposals have contributed to establish a solid and wide theoretical basis in this field. However, the existing tool support does not live up to the expectations mainly due to the complexity of putting this theory into practice.

In order to improve this situation, this thesis proposes the use of the Model Driven Development paradigm as a way to handle this complexity. Thereby, it first defines a methodological framework that advocates for the use of meta-modeling and model transformation techniques to tackle the design and implementation of software production methods. Then, this thesis introduces a software architecture that establishes the set of components that are required to support the methodological framework and gives implementation details of this architecture in the context of Eclipse, more specifically on the MOSKitt platform. The developed prototype has been called MOSKitt4ME.

Finally, in order to validate the proposed methodological framework and software architecture, the MOSKitt4ME prototype has been used for the development of a case study. This case study consists of a software production method that defines a Model Driven Development approach for the generation of web applications supporting business process specifications. The MOSKitt4ME prototype has successfully supported the design of the case study and the construction of its supporting CASE tool.

Resumen

La Ingeniería de Métodos surgió como disciplina hace dos décadas con el objetivo de afrontar el reto de dar soporte a la definición de métodos de producción de software, su adaptación a las necesidades de proyectos específicos y la construcción de las correspondientes herramientas CASE de soporte. Durante estos veinte años muchas propuestas teóricas han contribuido a establecer una base teórica amplia y sólida en este campo. Sin embargo, las herramientas existentes no están a la altura de las expectativas principalmente debido a la complejidad que conlleva poner esta base teórica en práctica.

Con el objetivo de mejorar esta situación, esta tesis propone el uso del paradigma de Desarrollo de Software Dirigido por Modelos como solución que permite manejar esta complejidad de forma eficiente. De este modo, primero se define un marco metodológico que hace uso de técnicas de meta-modelado y transformaciones de modelos para abordar el diseño e implantación de métodos de producción de software. Además, se presenta a continuación una arquitectura software que establece el conjunto de componentes que permiten dar soporte al marco metodológico y se proporcionan detalles de implementación de la arquitectura propuesta en el contexto de Eclipse, más concretamente de la plataforma MOSKitt. El prototipo desarrollado se ha llamado MOSKitt4ME.

Por último, a fin de validar el marco metodológico y la arquitectura software propuestas, el prototipo MOSKitt4ME ha sido usado para el desarrollo de un caso de estudio. En concreto, este caso de estudio consiste en un método de producción de software que define una propuesta basada en el Desarrollo de Software Dirigido por Modelos para la generación de aplicaciones web de soporte a procesos de negocio. El prototipo MOSKitt4ME ha proporcionado un adecuado soporte al diseño del caso de estudio y a la construcción de la herramienta CASE de soporte.

Acknowledgements

First and foremost, I want to express my most sincere gratitude to my supervisors Vicente, Manoli and Victoria, for giving me the opportunity to undertake this fascinating work. Without your deep research expertise and continuous support, this work would not have been possible. This thesis is as much yours as it is mine.

A big gratitude is also due to Isma, Miriam, Nacho, Pablo, María, Clara, Pau, Arthur, Ainoha and the rest of colleagues from the ProS research center for all the good moments we have shared together. These moments have been of incalculable help.

I also want to give special thanks to Laura, because behind every researcher there is always someone with unlimited patience. Thanks for all the love and motivation you have given me.

Finally, I want to thank the *Conselleria de Infraestructuras y Transporte* for the research fellowship that has provided all the economic resources needed to make possible the development of this work.

Contents

List of Figures

List of Tables

1. Introduction

Software development projects have proven to be highly diverse in nature due to the wide variety of situational elements that come into play. These elements, especially those regarding human and organizational factors, have a great impact on the development process. Therefore, in order to maximize productivity and improve the quality of the developed software, the development process must be governed by a software production method that is adapted to these situational needs.

This fact has already been acknowledged in other works such as [25], [26] and [41]. For instance, in [26] it is stressed that general-purpose methods[1] are "weak" in the field of problem solving compared to the solutions adapted to the problem at hand. In order to cope with this need for method adaptation, it is necessary to find alternatives that not only enable the in-house definition (and adaptation) of methods but also the construction of the corresponding supporting tools. Up to now, the Situational Method Engineering discipline seems to be the most promising solution to supply this need.

The Situational Method Engineering discipline encompasses all the aspects regarding the creation of methods for specific situations [41] and constitutes a sub-area of a broader field called Method Engineering. Method Engineering is defined in [9] as *the engineering discipline to design, construct and adapt methods, techniques and tools for the development of information systems*.

During the last two decades, a lot of proposals have tried to provide an answer to the existing problems in this area. However, while these proposals have contributed to establish a solid theoretical basis, none of them has been successfully exploited in industry, being relegated just to educational

[1] In this document, the terms "method" and "methodology" are used as synonyms of software production method

environments. In order to turn Method Engineering into reality, this master's thesis provides a methodological approach that covers the main phases of the Method Engineering lifecycle[2] from a Model Driven Development (MDD) perspective. In particular, the proposal advocates for the use of meta-modeling and model transformation techniques to tackle the design and implementation of project-specific software production methods.

The rest of this chapter is organized as follows: First, section 1.1 presents a motivation of the work described in this thesis. Then, section 1.2 states the problem that this thesis tackles and section 1.3 briefly presents the solution proposed to face this problem. Section 1.4 explains the context of this work and, finally, section 1.5 outlines the structure of the thesis.

1.1. Research Motivation

Software production methods guide software engineers during the course of software development projects by establishing the rules and procedures that assist in the orderly project execution. Thereby, methods define *what* to do, *how* and *when*, contributing to a better understanding of the problem and, therefore, to an improvement in quality of the developed software.

Even though different attempts have been made to develop universally applicable methodologies that fit any situation (e.g. Extreme Programming [6], the Rational Unified Process [47], etc.), real software development projects have demonstrated that methods must be tailored to fit specific context needs [26]. As a result, the "one-size-fits-all" methodology is now considered unattainable [22, 24, 26, 41, 88, 91]. In order to face this tailoring process, alternatives that support the definition of project-specific methodological approaches and the construction of supporting tools need to be sought. As stated above, the Situational Method Engineering discipline has emerged as the most optimistic solution to supply this need.

[2] In general, the Method Engineering lifecycle comprises the specification of the method requirements, the method design, the method implementation (i.e. the construction of the tool that supports the method) and the method validation [57]. The requirements analysis and validation of the method fall out of the scope of this thesis.

Within the (Situational) Method Engineering field, method engineers mainly focus on designing methods and implementing the tools that support such methods similarly to the way Information System Development (ISD) groups design and implement information systems. Method engineers can therefore be considered as developers of information systems for ISD [89]. Thus, the goal of Method Engineering is to improve the ISD process by providing better methods and supporting tools.

Facing the accomplishment of this goal is, however, a complex and error-prone task that requires automated tool support [57]. Unfortunately, while the theoretical basis that lays the foundations of Method Engineering is very solid and extensive, 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 tool support, namely Computed Aided Method Engineering (CAME) environments, aims at supporting the method engineering tasks but, nowadays, it mostly represents incomplete prototypes that present important deficiencies. This problem also becomes evident in [57] where a study of different CAME environments is presented.

In view of this situation, it is apparent that there is a significant need for software tools that provide appropriate support to Method Engineering. Since the development of this kind of tools is very far from trivial, it is crucial to find engineering solutions to properly handle this complexity. The research of these solutions has constituted the central focus of this work, and the obtained results are presented in this thesis.

1.2. Problem Statement

The proper performance of tasks such as the design and implementation of software production methods is particularly difficult without the assistance of appropriate tools. The discussion presented in the previous section illustrates this fact. Nevertheless, tool support still remains the Achilles' heel of Method Engineering and this handicap is leading to a slow industry adoption of Method Engineering approaches [80].

In order to improve this situation, the work presented in this thesis contributes to the Method Engineering field by tackling from a MDD perspective the following two challenges:

Challenge 1. Definition of a methodological approach that establishes the series of well-defined steps that allow method engineers to systematically define methods and build the corresponding supporting tools. To support these tasks in an effective manner, the approach must be based on a sound infrastructure that formalizes:

- **Req. 1.1.** The concepts that are available for defining methods and the rules governing their use.
- **Req. 1.2.** How the method specifications are created using the formalized concepts.
- **Req. 1.3.** The mechanisms that enable the definition of mappings from method specifications to the CASE tools that support them.

Challenge 2. Definition of an architecture that establishes the collection of components (and the interaction between these components) that must be implemented in a software tool in order to support the various phases that compose the methodological approach. This architecture must fulfill the following non-functional requirements:

- **Req. 2.1.** Technology-independence. The architecture must be defined in a technology-independent fashion in order to make it less sensible to technological changes and facilitate its implementation in different platforms.
- **Req. 2.2.** Modularization. The architecture must be based on separate components in order to improve its maintainability and facilitate its evolution.
- **Req. 2.3.** Separation of concerns. The architecture must clearly separate components that deal with Method Engineering tasks (e.g. method specification) from components that deal with ISD tasks (e.g. system specification).

1.3. Proposed Solution

This section briefly summarizes the solutions proposed in this master's thesis to face the challenges stated above.

First of all, regarding **challenge 1**, this thesis defines a methodological framework that defines the method, languages and techniques that allow method engineers to perform in a systematic way the design and implementation of project-specific software production methods. This methodological framework is built upon an MDD infrastructure [4] that lays the foundations of the framework and is based on meta-modeling and model transformation techniques. On the one hand, the meta-modeling techniques are based on the Software & Systems Process Engineering Meta-model (SPEM) [87] (**req. 1.1**) and are the means that allow the method engineer to produce method specifications as machine-processable models (**req. 1.2**). On the other hand, model transformations make use of these models for (semi)automating the performance of the method implementation (**req. 1.3**).

By applying these ideas, it has been possible to define a methodological framework that not only tackles the definition of methods following a widely accepted standard, but also proposes to use these definitions for the (semi)automatic generation of CASE tools that integrate all the required elements to provide rich support to the methods (from simple textual editors to more sophisticated tools such as graphical editors, code generators, report generators and process engines).

Furthermore, regarding **challenge 2**, this thesis also proposes an architecture that specifies the technology-independent components (**req. 2.1 and 2.2**) that are needed to support the methodological framework. This architecture is divided into two parts (**req. 2.3**): on the one hand, it defines the components that must be implemented in a CAME environment so that it enables the definition of methods and the (semi)automatic generation of CASE tools. On the other hand, it defines the various components that are included in the CASE tools that are obtained by means of the CAME environment. The definition of these components is necessary in order to establish the transformation mappings between the method models and the CASE tools.

Finally, as a proof of concept of the proposed solution, this thesis provides implementation details of a CAME environment that is being developed in the context of Eclipse, more specifically in the context of the MOSKitt platform [55]. This tool is based on the defined architecture and provides support to the methodological framework.

1.4. Context of the Thesis

This master's thesis has been developed in the research center *Centro de Investigación en Métodos de Producción de Software* (ProS) of the *Universidad Politécnica de Valencia*. More specifically, the solutions proposed in this work have been defined and implemented within the context of the MOSKitt project [55].

The MOSKitt project constitutes a jointly work developed by the *Conselleria de Infraestructuras y Transporte* (CIT) and the *ProS* to develop an Eclipse-based CASE tool to support the *gvMétrica* method (an adaptation of *métrica III* to satisfy CIT needs). There is a big community involved in the project, ranging from analysts (software and business analysts) to end users, which are in charge of validating each new release of the tool. This setting constitutes an adequate environment to validate the proposal presented in this thesis. In fact, in the near future the results of this work will be included into a MOSKitt released version in order to use it for the definition of *gvMétrica* and the construction of the supporting tool.

1.5. Outline

The remainder of this thesis is organized as follows:

- Chapter 2 presents a study about the current state of the art, focusing on Method Engineering approaches, languages and tools. Specifically, this study stresses the limitations of the works that are presented and details how the proposal described in this thesis tackles these limitations.

- Chapter 3 presents in detail the methodological framework that is proposed in this thesis for performing the design and implementation of software production methods.

- Chapter 4 defines the architecture that establishes the software components that are required to support the methodological framework presented in chapter 3. Furthermore, this chapter introduces a prototype that has been developed on the MOSKitt platform in order to implement the proposed architecture.

- Chapter 5 describes the case study that has been chosen to validate the proposal. Furthermore, it details how this case study has been developed on the prototype presented in chapter 4.

- Chapter 6 presents the conclusions of this thesis and outlines future work that can be carried out in order to extend the proposal. Furthermore, this chapter lists the research publications that have been produced during the course of this work.

2. State of the Art

The term *Method Engineering* was first introduced in the mid-eighties by Bergstra et al in [7], and was later used in other works such as [9], [48] and [85]. From that moment, Method Engineering emerged as a promising way to tackle the adaptation of software production methods and tools to specific project needs.

Since the origin of Method Engineering two decades ago, this discipline has had an extensive history. Many works developed both at academia and industry have contributed to establish a solid theoretical basis in this field. In order to underpin this theory, a survey of the most relevant contributions is gathered in [41].

Specifically, this chapter analyzes some of the most important Method Engineering proposals, addressing three topics: (1) Method Engineering approaches, (2) languages for building method specifications and (3) software tools supporting these approaches and languages. According to these topics, section 2.1 first presents different approaches for method definition. Then, in section 2.2, some of the most significant languages that have been proposed in the literature to perform this definition are described. Section 2.3 surveys some tools that have been developed to support the approaches and languages previously presented and, finally, section 2.4 draws some conclusions.

2.1. Method Engineering Approaches

Many Method Engineering approaches of different nature have been proposed during the last two decades. For instance, approaches such as [11] or [63] tackle method construction as an assembly of method components. Others, however, focus on the spreading and sharing of methodological knowledge rather than the definition and adaptation of methods. This is the case of the community based approach proposed in [53], which aims at solving method usage problems by improving the practitioners' understanding of the method to apply. Furthermore, proposals such as [15] propose the use of patterns for performing method extensions while others such as [32] and [43] offer a service-oriented view of Method Engineering.

In view of this disparate scenario, this section aims to provide a survey of the most extended types of approach. In particular, these types have been classified according to the types proposed in [69], which are: (1) the assembly-based approach, (2) the paradigm-based approach and (3) the extension-based approach. In order to illustrate the steps that must be followed to perform each of these approaches, the *Map* process meta-model proposed in [79] is used. This meta-model enables the creation of intuitive process models based on the notions of *intentions* to fulfill and *strategies* to achieve these intentions.

2.1.1. The assembly-based approach

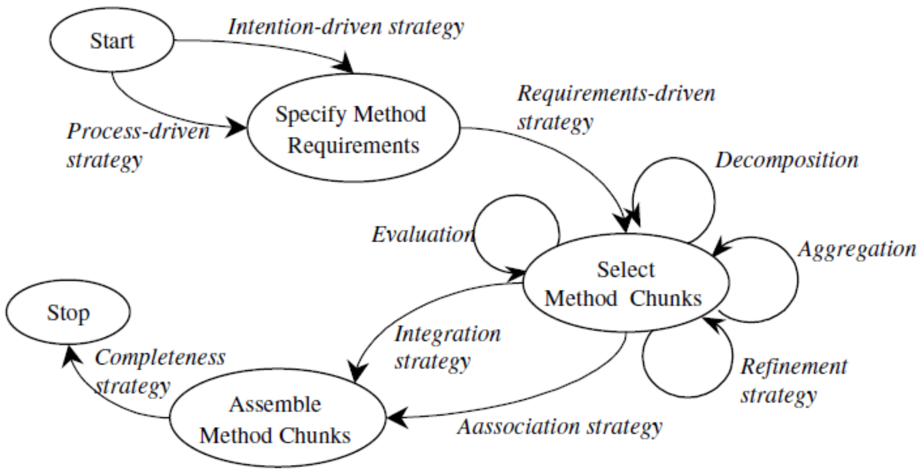


Fig. 2.1. Assembly-based approach (from [69])

The assembly-based approach [67, 69] consists in the construction of software production methods by means of the assembly of reusable method chunks (or fragments) that are stored in some method base repository [10, 35, 66]. Thus, methods are viewed as a collection of chunks that are "glued together" to form a method attuned to specific context needs. In order to show the different steps that must be followed to perform the method assembly, Fig. 2.1 shows this approach as a process model following the map notation.

The assembly-based approach is the most common of the three approaches. This is mainly due to the fact that this approach advocates for a modular vision of methods, which entails important advantages. Between these advantages, reusability is of high significance. Specifically, a modular vision of methods facilitates the reusability of their different parts, which directly leads to a reduction of the time required to define new methods. Furthermore, considering a method as an assembly of components also has a positive impact on its evolution qualities, such as maintainability and extensibility.

Some examples of relevant Method Engineering proposals that follow this approach are Brinkkemper's [9, 10, 11] and Prakash's [63]. On the one hand, Brinkkemper mainly focuses on method fragment assembly techniques and their formalization by means of first-order logical formulas. In [11] he stresses the need of imposing constraints in the assembly process in order to obtain meaningful methods. Most of the constraints that he proposes are syntactical, but he emphasizes the need of defining semantical constraints as well, which requires the formalization of the fragment semantics. He carries out this formalization by means of an ontology.

On the other hand, Prakash proposes an approach to formal method specification. This approach is based on three levels: the generic view (the most abstract view of a method, independent of the underlying paradigm), the meta-model and the method (obtained by instantiating the meta-model). These three layers represent an attempt to develop a comprehensive framework and architecture for methodology domain modeling. Specifically, in this approach a method is viewed as a collection of *method blocks*, which are defined as pairs <objective, approach>. The objective of a method block establishes what the block tries to achieve and the approach defines the technique that can be used to achieve the objective of the block. In addition, he also proposes different types of blocks, such as *product manipulation* and *constraint enforcement* (for atomic methods), and *product composition* and *compositional-constraint enforcement* (for compound methods).

The nomenclature problem

In the Method Engineering literature, proposals that follow the assembly-based approach denote the atomic element from which methods can be assembled in different ways. For instance, Ralyté uses the term method chunk, while Prakash uses the term method block and Brinkkemper the term method fragment. In order to reach a consensus on the definition of this atomic element, in [39] a study of the different terms that have been proposed is presented. In general, the most accepted are *method fragment* and *method chunk*.

On the one hand, method fragments can be either *product fragments* or *process fragments* [9, 35, 65, 76]. In general, a product fragment describes a product that is either consumed or produced during the method. A process fragment describes activities and procedures that must be executed to construct products. On the other hand, method chunks [52, 66, 67] can be defined as the combination of a process fragment and a product fragment.

During the last decade, there has been much debate about the efficacy of a method chunk as compared to a method fragment. While method chunks offer some advantages, it seems that method fragments are quite more flexible. For instance, one advantage of method chunks is argued to be the speed of usage, since a smaller number of chunks is usually required to assemble a complete method. However, there is a potential disadvantage as a result of the fact that the process-product linkage present in method chunks is neither one-to-one nor unique in real-life scenarios. Thus, the separation between product and process that method fragments provide implies important advantages such as the possibility to relate one process fragment with many product fragments and the possibility to reuse one product fragment in the definition of many process fragments [39].

Specifically, in the proposal presented in this thesis the concept of method fragment is used. One of the reasons for this is the language used in the proposal, i.e. the SPEM standard (see section 2.2.6). In particular, the separation of product and process fragments allows method engineers to leverage the clear separation between method product and process provided by SPEM.

2.1.2. The paradigm-based approach

The paradigm-based approach [69, 71] is based on some initial idea expressed as a model or a metamodel that is called the *paradigm model* and supports the evolution of this paradigm model into a new model satisfying another engineering objective. In other words, the hypothesis of this approach is that a new method is obtained either by abstracting from an existing model or by instantiating a meta-model. Thereby, this approach uses meta-modeling as its underlying Method Engineering technique.

One of the results obtained by the meta-modeling community is the definition of any method as composed of a product model and a process model [64]. A product model defines a set of concepts, their properties and relationships that are needed to express the outcome of a process. A process model comprises a set of goals, activities and guidelines to support the process goal achievement and the action execution. Therefore, method construction following the meta-modeling technique is centered on the definition of these two models [71]. This is illustrated in Fig. 2.2, wherein a map representing the paradigm-based approach is shown.

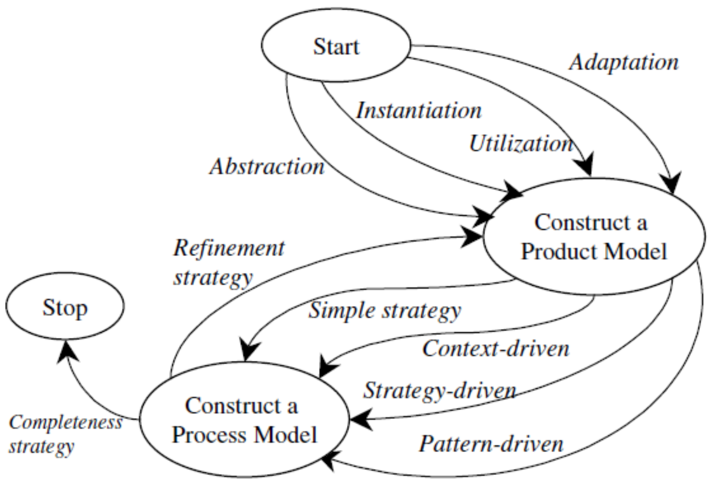


Fig. 2.2. Paradigm-based approach (from [69])

The paradigm-based approach is the most generic of the three approaches presented in this survey. Since it is based on meta-modeling, it presents important benefits that are inherited from this technique. For instance, since methods are defined at a high level of abstraction, their understandability is increased (compared to, e.g, textually defined methods), thus contributing to facilitate their application in real ISD projects.

Some examples of relevant Method Engineering proposals that follow this approach are Rolland's [75] and Grundy's [30]. On the one hand, Rolland presents in [75] a proposal for defining *ways-of-working* in a systematic manner. A way-of-working is a process model that takes into account heuristic knowledge to guide humans performing systems development. Specifically, these process models are created by instantiation from a process meta-model that is called NATURE (see section 2.2.5). Furthermore, the product meta-model presented in [82] enables the definition of the product part of these models.

On the other hand, Grundy [30] proposes a product-oriented approach from defining methods. Specifically, he defines a product meta-model called CoCoa that allows method engineers to define the design notations that enable the creation and manipulation of the method products. Furthermore, a process modeling environment called Serendipity is proposed for supporting the definition of process models that coordinate the development of the method products.

2.1.3. The extension-based approach

The extension-based approach [69] consists in identifying typical extension situations and performing the required extension of the method by means of extension patterns. A pattern is a component that describes a recurrent problem [15], which helps to identify the extension situation, and is defined with its associated solution (the guidelines to be followed when the pattern is applied). Specifically, this solution embodies the process chunk that is to be applied on a particular product [15].

In view of this definition of the pattern concept, one may think that the process of extending a method is somewhat similar to the assembly of method fragments. Actually the main difference lies in the nature of the components that participate in the assembly or the extension. In the former case, the components (i.e. the method fragments/chunks) can be directly used, whereas in the latter they cannot, that is, they have to be generated from the generic patterns.

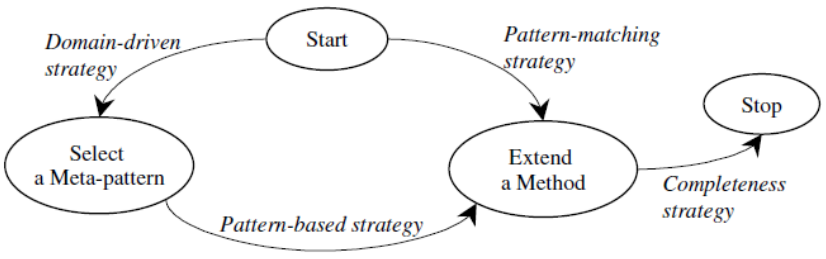


Fig. 2.3. Extension-based approach (from [69])

Fig. 2.3 shows the map representing the process underlying the extension-based approach. Even though this approach is the less common of the three approaches, it also presents important advantages. For instance, it is oriented towards guiding the method engineer during the performance of method extensions, which means that it is an adequate approach for performing the adaptation of methods to context needs, one of the main goals of Method Engineering.

An example of proposal that suggests the use of patterns for performing method extensions is [15]. Specifically, this work proposes a set of generic patterns for introducing temporal features (such as time constraints) to object oriented models.

2.1.4. General discussion

Method Engineering has a disparate history since many different approaches have been proposed over the past twenty years. However, all these approaches share a common goal: assisting the method engineer during the definition of methods and their adaptation to the context needs. In order to reach this goal, most proposals advocate for the assembly-based approach, but others promote other approaches, such as the paradigm-based or the extension-based. Together, all these proposals have contributed to establish a solid and wide theoretical basis in the area of Method Engineering. However, in spite of this sound basis, **there still remains a need for a Method Engineering proposal that takes all the method dimensions into account together**. Currently, most of the Method Engineering proposals only focus on the *product* dimension (the products to be constructed during the method) and the *process* dimension (the process to be followed to obtain the products), but other dimensions are also important. These dimensions are basically the *tool* dimension (the software tools that provide support to the *product* and *process* dimensions) [83], and the *people* dimension (the agents that make use of the *tools* in order to develop the method *products* following the method *process*) [41].

This problem has already been noted in other works such as [43]. In order to fill this gap, the methodological framework presented in this master's thesis covers all these four dimensions of methods, as will be shown in chapter 3.

2.2. Method Engineering Languages

Meta-modeling is considered by the Method Engineering community as "the core technique in Method Engineering" [69] as it provides an effective way to formalize the abstract syntax of the language that establishes the concepts, constraints and rules that are applicable in the construction of the software production methods. In particular, this subsection presents a survey of some of the most significant languages that have been proposed during the last two decades. For each of these languages a brief overview is given, and later in section 2.3, tools supporting them are described.

2.2.1. ASDM

The semantic data model notation ASDM [42] is a forerunner of current Method Engineering languages and yet it provides a powerful means for representing ISD knowledge. Furthermore, it represents the first attempt to define method semantics, as noted in [56]. An overview of the meta-model is presented in Fig. 2.4.

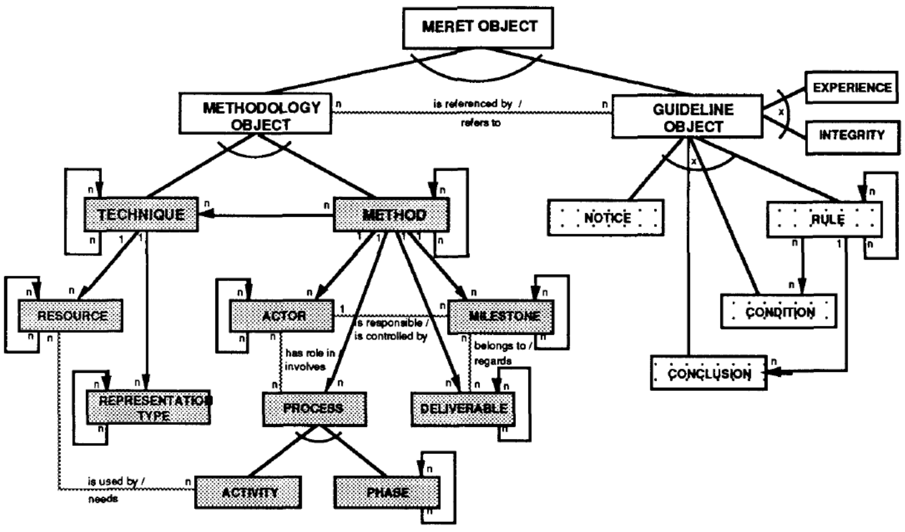


Fig. 2.4. ASDM meta-model (from [42])

As shown in Fig. 2.4, the *MERET Object* is the most general object type. A MERET object can be either a *methodology object* or a *guideline object*. On the one hand, methodology objects embrace all description objects for the specification of a method: *techniques* (used to develop products), *actors*, *milestones*, *processes*, etc. Processes can be either *phases* or *activities* (elementary units of work). On the other hand, guideline objects reflect the more dynamic part of the method knowledge. For instance, a guideline can be a textual *notice* representing experiences from applying a specific part of the method, or an integrity *rule* represented as a horn clause.

In general, the ASDM meta-model provides adequate concepts for specifying software production methods in a product-oriented fashion. Furthermore, it provides a graphical notation that facilitates method comprehension. However, it just embodies a first step towards Method Engineering since it presents important deficiencies. For instance, it provides poor support to the specification of the process part of methods, which negatively affects the possibility of building complete CASE environments from ASDM method specifications.

2.2.2. GOP(P)RR

The GOPRR conceptual data model [46] is a Method Engineering language that has been specially designed to support the definition of techniques that can be used for the manipulation of the method products. In other words, it supports the definition of modeling languages, such as ER, DFD, UML Class Diagram, etc.

The name GOPRR is an acronym that stands for the metatypes the language operates on: *Graph*, *Object*, *Property*, *Role* and *Relationship*. This metatypes and their relationships are graphically illustrated in Fig. 2.5.

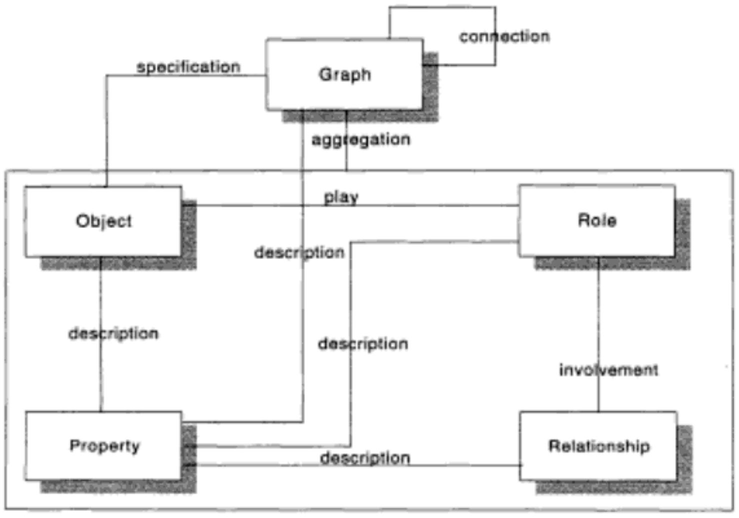


Fig. 2.5. GOPRR meta-model (from [34])

In particular, these concepts represent the following:

- Graph: A graph is a collection of objects and relationships among these objects via roles. An example of graph is a UML class diagram.
- Object: An object is an element that can be placed on its own in a graph. An example of object is a Class that belongs to an UML class diagram.
- Relationship: A relationship is an explicit connection between two or more objects. Relationships attach to objects via roles. An example of a relationship is an Association of a UML class diagram.

- Role: A role specifies how an object participates in a relationship.
- Property: A property is a describing or qualifying characteristic associated with the other types. An example of property is an Attribute that belongs to an UML class diagram.

Furthermore, the notion of *Port* is included in the GOPPRR language [51], which represents an evolution of the GOPRR language. Specifically, a port is an optional specification of a specific part of an object to which a role can connect. Normally, roles connect directly to objects, and the semantics of the connection are provided by the role type. If you want a given role type to be able to connect to different places on an object with different semantics, you can add ports to the object's symbol.

To summarize, the GOPRR and GOPPRR languages represent an adequate means for defining modeling notations that can be later used in an integrated CASE environment for creating and manipulating method products. However, this approach presents important lacks. While the CASE tools obtained from GOP(P)RR specifications may provide complete support to the manipulation of method products, they overlook important aspects of ISD such as process enactment and code generation.

2.2.3. MEL and MDM

The Method Engineering Language (MEL) [12] is a formal representation language that provides concepts and constructs for the textual description, selection and manipulation of method fragments. On the one hand, it provides syntactic constructs to compose from activities complex processes such as sequential execution, conditional branch, iteration, parallel execution and non-deterministic choice. On the other hand, it provides constructs for the detailed specification of the products that these processes need as input and deliver as output.

Furthermore, the semantic aspects of the product fragments can be specified by anchoring the fragment descriptions to an ontology, described by means of the Methodology Data Model (MDM) [35]. Anchoring means that a method fragment is described in terms of well-defined basic concepts and associations between those concepts. In particular, the MDM ontology provides the following concepts (CN_0) and associations (A_0):

- CN_0 = {Activity, Actor, Association, Attribute, Attribute Type, Benefit, Business Area, Channel, Communication Protocol, Condition, Cost, Critical Success Factor, Data Flow, Data Collection, Decision, Dialogue, Event, External Entity, Field, Function, Goal, Group, Location, Node, Object, Object Class, Opportunity, Organizational Unit, Problem, Requirement, Role, Rule, Solution, State, Strength, System, Threat, Transition, Weakness}

- A_0 = {Abstraction, Aggregation, Alternative, Balance, Base, Capability, Change, Choice, Component, Connection, Constraint, Consumer, Contents, Dependence, Description, Effect, Employment, Expression, ExternalOutput, Imposition, Input, Interaction, Involvement, Manipulation, Message, Output, Performance, Place, Price, Producer, Product, Request, Resource, Responsibility, Screen, Site, Specialisation, Support, TransitionTrigger, Trigger, Usage}

In summary, MEL can be considered as a complete Method Engineering language. It provides constructs for the definition of both products and process fragments, and also for their manipulation and assembly. Furthermore, it partially covers the method people dimension through predefined property types such as “creator” and “responsible”. Unfortunately, the high amount of properties and concepts make it hard to learn, and its textual nature hinders the understanding of the developed methods.

2.2.4. MRSL and MVM

The Method Requirements Specification Language (MRSL) [31] is a textual language for specifying method requirements in a technology-independent fashion. The main objective of this language is to enable method engineers to express method requirements in simple terms, avoiding the need to have expert knowledge about meta-models and how to instantiate them. These requirements can be later used to (semi)automatically obtain the final method specification and the CASE tool support.

MRSL is based on the Method View Model (MVM) meta-model, which is presented in Fig. 2.6. MVM method concepts are called *things*, and are partitioned into *product entities*, *links* and *constraints*. A link is any *thing* that connects two product entities together. Constraints are those *things* that can be used by software engineers to specify properties of links and product entities. Finally, any *thing* that is not a link or a constraint is a product entity.

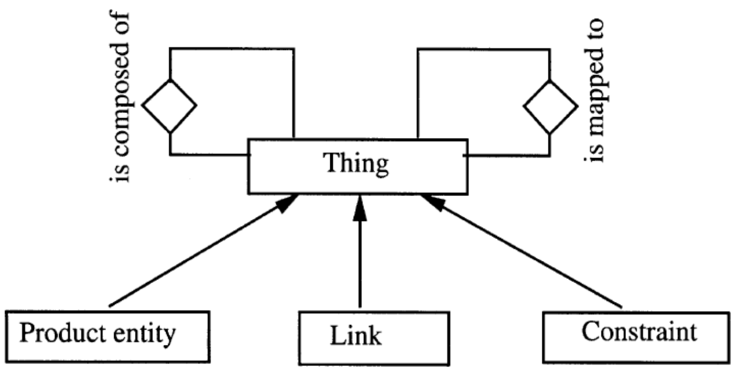


Fig. 2.6. MVM meta-model (from [31])

In particular, this language is similar to the GOPRR and GOPPRR languages, in the sense that it supports the definition of modeling languages such as DFD, ER, etc. but not the definition of complete software production methods. Therefore, the CASE environments that can be obtained from method specifications that follow this language overlook important aspects of ISD such as process enactment and code generation.

2.2.5. NATURE

The NATURE[3] modeling formalism [74, 77] consists of a set of generic concepts and their relationships for constructing methods from a process perspective, and was designed with a certain philosophy in mind: process models must be contextual, that is to say, the process model must allow users to switch context in a flexible and easy manner. Specifically, a context is composed of the *situation* that is perceived by the method engineer and the specific intention (or *decision*) he/she has in mind. The NATURE meta-model (see Fig. 2.7) addresses these issues by making the notions of situation, decision and context explicit.

[3] Novel Approaches to Theories Underlying Requirements Engineering

Thereby, the notion of *context* constitutes the basic building block of NATURE process models. Contexts are defined as couples <situation, decision> that can be linked repeatedly in a hierarchical manner to define *trees*. A tree represents a structured piece of knowledge for supporting decision making in the process. In other words, it is a process fragment which aims at assisting the method engineer in making the most appropriate decision for the situation at hand. Finally, a collection of trees (i.e. hierarchies of contexts) is referred to as a *forest*, which represents the *method*.

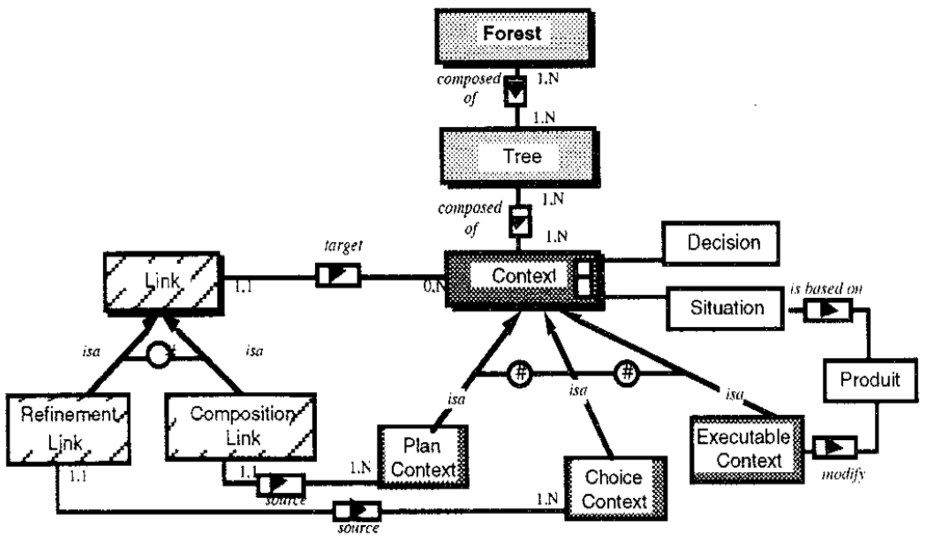


Fig. 2.7. NATURE meta-model (from [77])

To summarize, the NATURE approach represents a powerful means for specifying modular ISD processes that are easy to adapt to context changes and to assemble between each other to compose bigger processes. However, it must be used in combination with a product meta-model in order to specify complete software production methods. Furthermore, this approach does not support the *people* and *tool* dimensions of methods.

2.2.6. SPEM 2.0

In view of the diversity of Method Engineering languages that was emerging in the literature, the OMG[4] proposed the definition of a formal framework for the definition of software production methods and their components. The result is the standard language SPEM (Software & Systems Process Engineering Meta-Model) [87]. Specifically, this section focuses on its version 2.0, released on April 2008.

The SPEM 2.0 meta-model is structured into seven main meta-model packages as depicted in Fig. 2.8. In general, meta-model classes are introduced in lower packages as simply as possible, and then, they are extended in higher packages via the merge mechanism. By means of this mechanism additional properties and relationships can be added in order to realize more complex process modeling requirements.

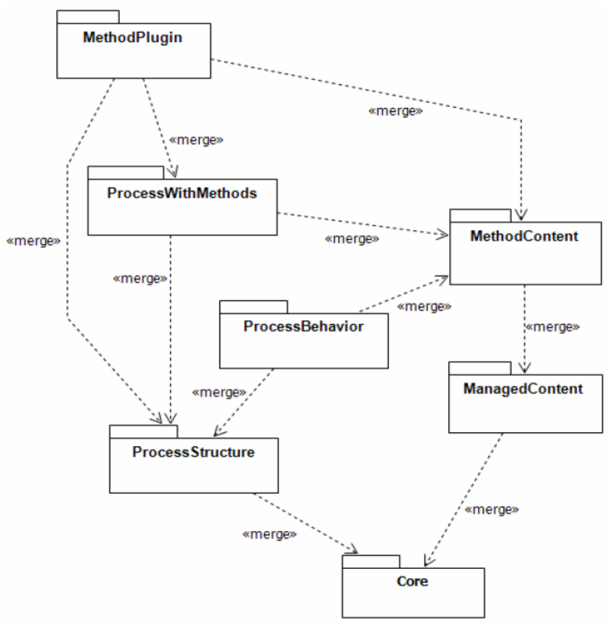


Fig. 2.8. Structure of the SPEM 2.0 meta-model (from [87])

[4] Object Management Group, http://www.omg.org/

- **Core:** This package contains the classes and abstractions that build the base for classes in all other meta-model packages.
- **Process Structure:** It contains the base classes for all process models. Specifically, a SPEM 2.0 process is represented by a breakdown structure composed of *Activities* that reference the performing *Role* classes and the input/output *WorkProduct* classes. Furthermore, it provides mechanisms for process reuse, e.g. process patterns.
- **Process Behavior:** This package contains the classes that enable the use of behavioral models for extending the static breakdown structures built by means of the Process Structure package. However, it does not define its own behavior modeling approach, but rather provides 'links' to existing externally-defined behavior models. For example, a process defined with the Process Structure concepts can be linked to UML 2 Activity diagrams that represent the behavior of such process.
- **Managed Content:** It contains classes for managing the textual documentation of processes (i.e. it enables the association of guidance elements with process structure elements). For instance, a SPEM 2.0 process can be comprised of a combination of instances of the *Guidance* class with a process structure using the relationships defined in this package.
- **Method Content:** This package defines the core elements of every method such as Roles, Tasks, and Work Products. Then, processes would reuse these method content elements and relate them into partially-ordered sequences that are customized to specific types of projects. As a result, SPEM methods provide a clear separation of method content definitions and development processes.
- **Process With Methods:** This package provides the classes that are needed to integrate (i.e. reference) method content into the processes defined using the Process Structure package. These classes can store the changes made to the method content classes that only apply in the specific process.
- **Method Plugin:** This package introduces concepts for managing maintainable, large scale, reusable and configurable libraries or repositories of method content and processes.

In general, SPEM represents an adequate language for Method Engineering since it not only covers the *product* and *process* dimensions of methods but also the other two, *people* and *tool* (by means of primitives such as *Role*, *RoleSet* and *ToolDefinition*.). Furthermore, it is oriented towards the modular definition of ISD processes, facilitating their assembly from existing parts (a characteristic that is directly related to the Method Engineering principles). However, industry adoption is being slow mainly due to the lack of supporting tools.

2.2.7. ISO/IEC International Standard 24744

The ISO/IEC 24744 [45] is an International Standard that defines a meta-model for the technology-independent specification of development methodologies in any area, although it is weighed towards software development methodologies. Its scope covers, inter alia, concepts such as *work units*, *work products*, *producers*, *stages* and *model units* (see Fig. 2.9). In addition to the meta-model, a graphical notation is provided to allow method engineers to represent complete methods using graphical constructs.

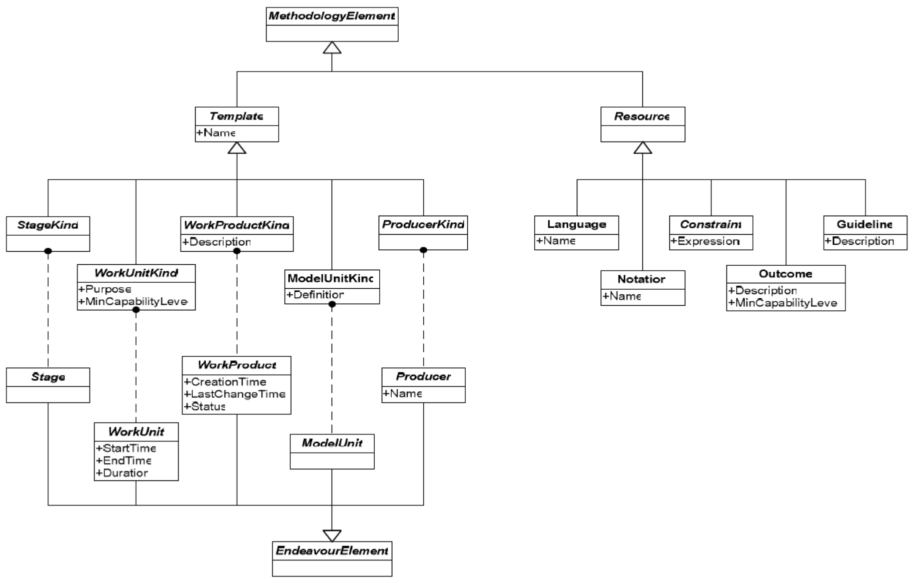


Fig. 2.9. Overall architecture of ISO/IEC 24744 (from [40])

One of the main novelties of the standard is the introduction of the powertype pattern concept of Odell [27, 28, 29, 40, 59] as a core element in the meta-model. Specifically, a powertype pattern is a pair of elements. The instances of one of them reside in the method domain and the others in the enactment domain. The elements in the enactment domain represent actual elements in use by the people on a particular project (e.g. actual tasks). On the other hand, the elements in the method domain represent method elements as they are specified in the model (e.g. instances of the meta-class Task). The main advantage of this approach is that it allows some attributes of the powertype to be inherited by the element in the method domain with values already allocated to them, while others remain "traditional" attribute specifications that get their value in the enactment domain.

In summary, this language embodies another standardization effort in the field of method definition. It represents a rather complete language for Method Engineering since it covers the *process*, *product* and *people* dimensions of methods. One of its distinctive characteristics is that it provides primitives for specifying elements that reside either on the instance level (i.e. enactment level) or the method level. This approach allows some attributes defined in the meta-model to be given values in any of both levels.

2.2.8. General discussion

This survey illustrates that there is a wide diversity of languages and all of them have their advantages and drawbacks. This conclusion is also drawn in several studies, such as [34] and [57]. These studies conclude that there is not ultimate Method Engineering language and, therefore, the choice of the language depends on the specific purpose and goals that one wants to achieve.

In order to contribute to improve this situation, standardization efforts are being made, e.g. the SPEM [87] and ISO/IEC 24744 [45] initiatives. These standards aim to provide languages dedicated to method specification that do not present the deficiencies found in previous proposals.

Unfortunately, while these standards represent adequate means to perform the definition of software production methods, **Method Engineering proposals that make use of these standards are still non-existent**. Indeed, in [41] it is predicted that one of the likely topics for research initiatives in the next years will be a new generation of CAME tools based on internationally standardized methodology meta-models.

In order to fill this gap, the methodological framework presented in this master's thesis proposes the use of the SPEM standard for the construction of the method specifications. Specifically, chapter 3 describes in detail how these specifications are built and how they are later used for the (semi)automatic generation of the CASE tool support. Then, chapter 4 gives implementation details of a CAME environment that supports these tasks.

2.3. Method Engineering Tools

The Method Engineering lifecycle is a complex and error-prone process that cannot be properly performed without automated tool support. The first Method Engineering supporting tools date back to the early days of Method Engineering, when the first academic prototypes were first introduced [57].

In general, there are two different types of tools: Computer Aided Method Engineering (CAME) environments [3, 35, 48, 78, 81] and MetaCASE tools [19, 44, 49, 50, 73]. Specifically this subsection presents, for each of these categories, a brief description and a survey of some of the most significant tools that have been developed during the last two decades.

2.3.1. Computer Aided Method Engineering (CAME)

Computer Aided Method Engineering (CAME) environments aim at supporting the definition of software production methods by means of languages such as NATURE or ASDM (see section 2.2). Thus, CAME environments are mainly focused on the method design phase of the Method Engineering lifecycle.

Fig. 2.10 illustrates the general architecture of CAME environments. As shown in the figure, CAME environments are made up of two parts: (1) the CAME part and (2) the CASE part.

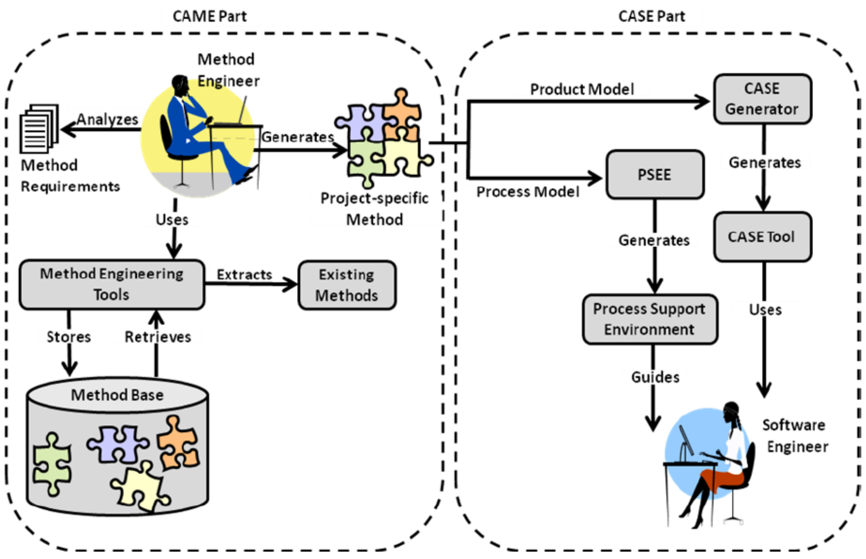


Fig. 2.10. General architecture of CAME environments (from [57])

On the one hand, the CAME parts offers facilities for method definition. Some examples of the functionalities that must be provided in this part are the following:

- Storage of method fragments/chunks in a repository (typically called Method Base).
- Definition of properties that enable the search and retrieval of method fragments/chunks from the repository.
- A query language for accessing the contents of the repository.
- Composition of method fragments/chunks.
- Support and guidance for the method engineer.

On the other hand, the main goal of the CASE part is to produce CASE tools and process support environments that enable the enactment of the method specified in the CAME part. For this purpose, the CASE part takes the method specification as input and offers means to manually or semi-automatically produce these tools.

MERET

The Methodology Representation Tool (MERET) [42] can be seen as the first approach towards a CAME tool for the specification, storage and further development of ISD knowledge. Specifically, it supports the specification of methods in a product-oriented fashion by means of the semantic data model ASDM, which is detailed in section 2.2.1. In addition, it addresses the integration of integrity rules and consistency checks on the method specifications.

One of the main drawbacks of this tool is that it only supports the specification of software production methods, lacking CASE tool generation capabilities. Furthermore, due to the product-oriented nature of ASDM, it provides poor support for defining the process dimension of methods.

Decamerone

Decamerone [33] is a CAME tool that provides facilities for specifying, storing and selecting method fragments, and for assembling them into a method. To perform these tasks, the tool provides the language MEL, described in section 2.2.3.

Fig. 2.11 shows the architecture of decamerone, which is divided into two parts: the CAME part and the CASE part. On the one hand, the CAME part is dedicated to the method design and contains the following components:

- **The user interface:** provides the required tools to perform the specification, selection and assembly of method fragments by means of the language MEL.
- **Method Base Management System (MBMS):** is the kernel of Decamerone. It provides the operations that are necessary to interact with the Method Base repository.
- **The MEL interpreter:** translates MEL specifications into MBMS function sequences.

On the other hand, the CASE part contains the required tools for the enactment of the method: a CASE tool repository, a process manager and a user interface that provides the editors that enable the system specification.

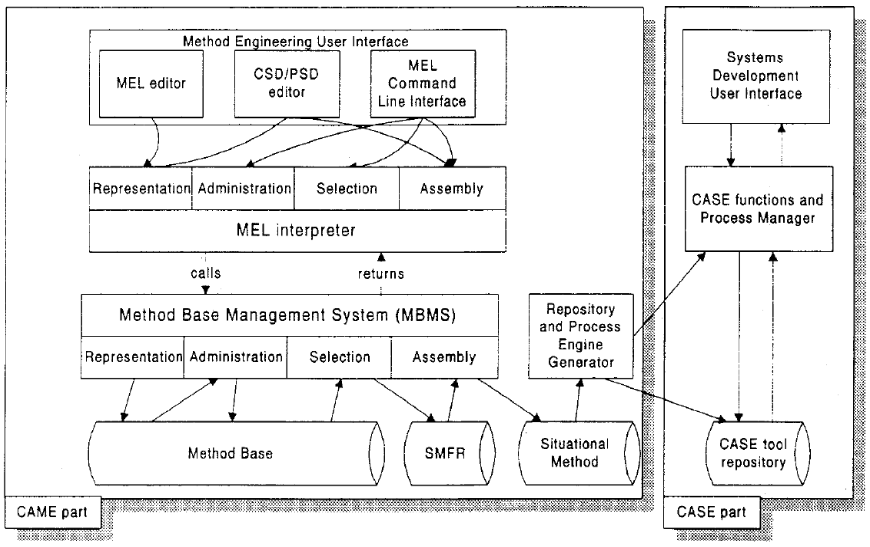


Fig. 2.11. Architecture of Decamerone (from [33])

In summary, Decamerone is a rather complete CAME environment. It supports the definition of methods by means of the MEL language, which not only allows the method engineer to define product and process fragments but also offers constructs for their manipulation (selection, storage, assembly, etc.). Furthermore, Decamerone supports the definition of the semantics of method fragments by means of the MDM ontology, and the generation of CASE tools that support both the product and process parts of methods. However, it also presents some deficiencies. For instance, it provides a poor graphical meta-model, and the textual nature of MEL complicates the understanding of the specified methods. In addition, the generated CASE tools lack code generation capabilities (i.e. the creation of method products by means of automatic tasks).

MENTOR

MENTOR [61, 84] is a CAME environment that aims at improving the productivity of method engineers by facilitating the construction of project-specific methods. It is based on the NATURE contextual approach, which is described in section 2.2.5.

Fig. 2.12 illustrates the architecture of MENTOR, which is composed of four main components:

- **The Method Engineering Environment:** this component contains viewers, editors and a generator. The viewers allow the method engineer to browse method fragments. The editors enable the graphical description of both product and process parts of methods. Finally, the generator aids in the automatic instantiation of predefined generic patterns.
- **The Application Engineer Environment:** this component represents the CASE part of MENTOR and contains the product editors that permit the development of the system specification. Furthermore, it contains a traceability tool that keeps track of product and process traces, and a process change manager that keeps coherent the element used during the process enactment when the process is modified.
- **The Guidance Engine:** this is the core component of MENTOR. It guides the method engineer in the performance of the Method Engineering tasks and enables the enactment of the specified process model.
- **The repository:** is structured in three levels that are interrelated: (1) the meta level, (2) the method level and (3) the workspace level. These levels contain respectively the product and process meta-models, method fragments, and process models and products under development.

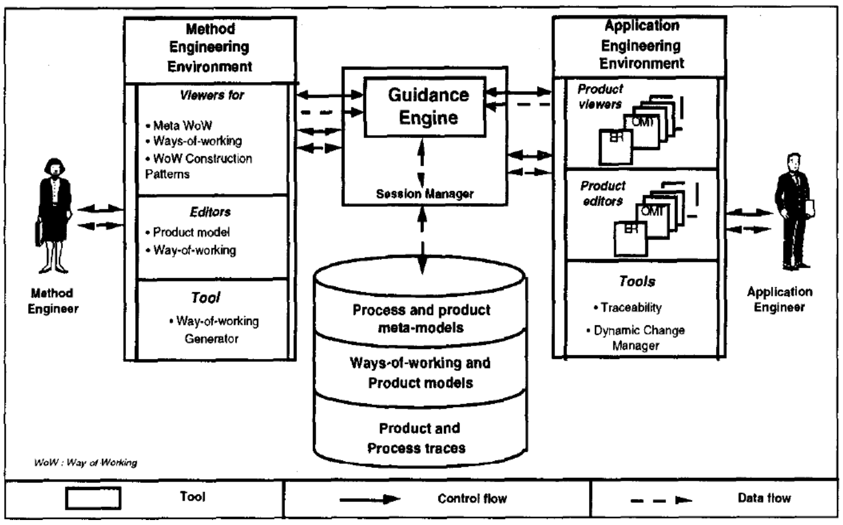


Fig. 2.12. Architecture of MENTOR (from [84])

To summarize, MENTOR is one of the most complete CAME tools. First of all, it supports the specification of method requirements, which is usually overlooked in most CAME environments. Furthermore, it supports both the assembly-based and paradigm-based approaches for method specification, and the construction of CASE tools that support the specified methods. However, it also presents some drawbacks. For instance, the generated CASE tools lack code generation, since they are only composed of product editors, a traceability tool and a process change manager. Furthermore, its graphical design is not intuitive, which negatively affects its usability.

Method Editor

Method Editor [81] is a CAME environment that uses UML as its meta-modeling language. In particular, the product part of methods is specified by means of the UML Class Diagram and the process part by means of the UML Activity Diagram. The specified methods are used by a *diagram generator* and a *navigator generator*. These tools perform the generation of the CASE environment that supports the method. This CASE tool is composed of a series of diagram editors that enable the creation and manipulation of the method products and browsing pages that guide the software engineer through the enactment of the method process.

In summary, Method Editor is one of the few CAME environments that support standard techniques such as UML. It is rather complete since it supports the specification of methods and the generation of CASE tools. Moreover, it provides a very intuitive graphical design. However, the main drawback of Method Editor is that, even though it supports CASE tool generation, these CASE tools only contain graphical editors that enable the creation/manipulation of the method products, and navigation pages that guide through the method process. Other aspects such as code generation or consistency checkers should be considered.

2.3.2. MetaCASE

Traditional CASE tools provide support to a single software production method. However, one fixed method simply cannot work for all software development projects and organizations as they differ significantly from one another and evolve over time [50]. Therefore, CASE environments should be adapted to meet the context needs, but this is not possible because the tools that support the methods are “hard-coded” in the environment.

The metaCASE technology aims at solving this problem. To achieve this goal, metaCASE tools add an additional level above the method level (see Fig. 2.13) in order to provide the ability to specify at a high level of abstraction the tools that are required to support the method, and then generate the CASE environment from these specifications.

As a result, unlike CAME environments (which are focused on the method design), metaCASE tools concentrate on the CASE tool construction (the method implementation phase of the Method Engineering lifecycle).

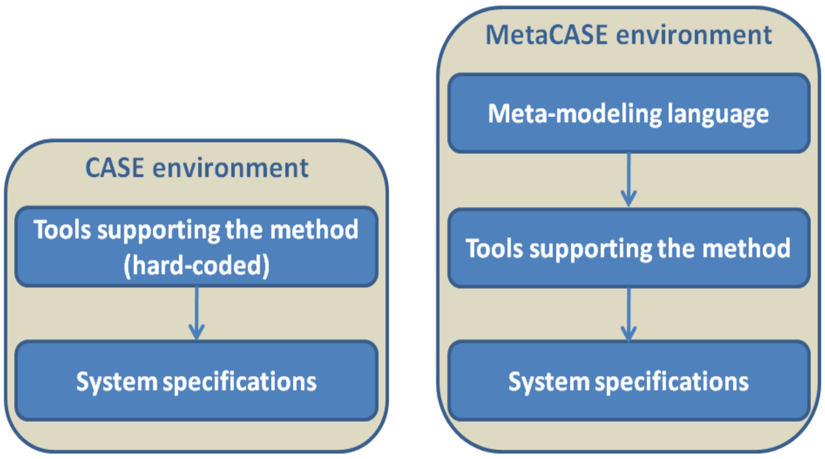


Fig. 2.13. CASE tool versus metaCASE tool

MetaEdit+

MetaEdit+ [46] is, up to our knowledge, the only Method Engineering tool that has been commercialized. It is a metaCASE environment based on the conceptual data model GOPPRR, described in section 2.2.2. By means of this language, MetaEdit+ enables the specification at a high level of abstraction of the modeling languages (in MetaEdit called “methods”) that have to be supported by the CASE tool under construction.

MetaEdit+ consists of several tool families. In particular, the family of tools that enable the specification of methods is the *Method Management Tools* family (see Fig. 2.14). This family is composed of the following tools:

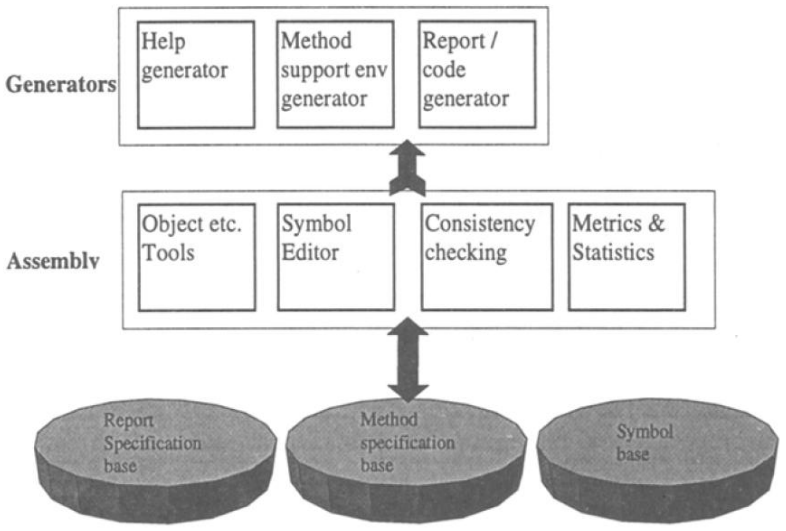


Fig. 2.14. Method Management Tools in MetaEdit+ (from [46])

- **The Method Base:** this repository stores method fragments and the symbols used for representing object types.
- **The Method Assembly System:** consists of the specialized tools that are needed for method assembly, such as *meta-model editors*, which allow the method engineer to specify methods by means of the GOPPRR language.
- **The Environment Generation System:** this system consists of the generators that take as input the method specifications and deliver the CASE tools.

In general, MetaEdit+ embodies an efficient solution for defining your own modeling languages. It is easy to use, well documented and has an intuitive graphical design. However, MetaEdit+ (and all metaCASE environments in general) falls short in providing adequate support to Method Engineering. This is due to the fact that these tools are focused on supporting CASE tool construction and overlook one fundamental aspect of Method Engineering: the definition of software production methods.

MERU

The Method Engineering Using Rules (MERU) metaCASE environment [31] offers the method engineer the textual language MRSL, which is built upon the MVM meta-model (see section 2.2.4). This language allows the method engineer to specify the method requirements in a technology-independent fashion. The document that is produced is called *Method Requirements Specification* (MRS) and is used to (semi)automatically obtain the final method specification and finally the CASE tool support.

MERU, like MetaEdit+, embodies an adequate tool for supporting the definition of modeling languages. Specifically, it provides a high number of features, such as process enactment support and method requirements specification. However, it lacks the possibility to specify software production methods that can assist the execution of real ISD projects.

2.3.3. General discussion

The survey presented in this section shows that in many Method Engineering initiatives CAME and metaCASE tools are developed in order to provide software support to their proposals. However, CAME and metaCASE technology is still immature, since most of these environments just represent incomplete prototypes that have only been used for academic purposes [57]. The main problem with these tools is that, in general, **CAME and metaCASE environments provide inadequate coverage of the Method Engineering lifecycle**. The main reason for this is that, on the one hand, CAME environments generally focus on the method design and, on the other hand, metaCASE environments concentrate on the method implementation. That is to say, CAME tools usually provide rich ways to specify software production methods but offer limited (or non-existent) CASE tool generation capabilities. On the other hand, metaCASE tools provide adequate means for building CASE environments but lack the possibility to define software production methods that can be enacted in real projects.

In order to fill this gap, the methodological framework presented in this thesis equally encompasses the method design and the method implementation. Therefore, the CAME environment that has been developed to support the proposal not only provides means for performing the definition of software production methods, but also for (semi)automatically obtaining the CASE tool support.

2.4. Conclusions

The survey presented in this chapter illustrates that the Method Engineering literature is very extensive. In particular, some of the most significant Method Engineering approaches, languages and tools have been presented and the following shortcomings have been identified:

Shortcoming 1. There still remains a need for a Method Engineering proposal that takes all the method dimensions into account together (i.e. the *product*, *process*, *tool* and *people* dimensions). Most of the existing proposals cover the product and process parts of methods, but the tool and people dimensions are almost completely overlooked.

Shortcoming 2. Method Engineering proposals that make use of standards for method definition (such as SPEM and ISO/IEC 24744) are still non-existent.

Shortcoming 3. CAME and metaCASE environments provide inadequate coverage of the Method Engineering lifecycle. In general, CAME environments focus on the method design and metaCASE environments on the method implementation.

The methodological framework proposed in this master's thesis addresses these shortcomings. Specifically, it proposes the use of the SPEM standard for performing the method design (**shortcoming 2**). This standard adequately supports the definition of methods that cover the product, process and people dimensions. The tool dimension is covered by means of the use of technical fragments [35] (**shortcoming 1**). Furthermore, in order to equally encompass the method design and implementation (**shortcoming 3**), the framework is founded on an MDD infrastructure that is based on meta-modeling and model transformation techniques. The meta-modeling techniques enable the definition of methods as models, and the model transformations enable to (semi) automatically obtain CASE tools from these models.

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. [10, 37, 52, 69]) 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. [20, 30, 46, 73]) 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 [4]. 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 [87], which is also described in chapter 2 (section 2.2.6).

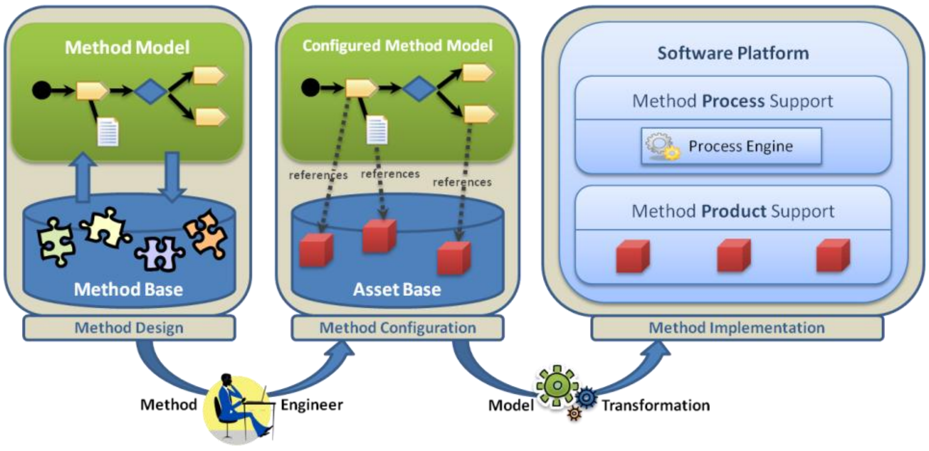


Fig. 3.1. Overview of the methodological framework

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*[5]. 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

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

following the RAS standard [72]. 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[6] 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.

[6] 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 [29], [36] and [38]. In general, proposals that focus on the method design phase usually use meta-modeling as their underlying technique to define the method specifications [11, 42, 52]. 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 [30, 46, 73].

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 [58]. 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, [58] 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 [21] method fragments are generated by instantiation from one of the top levels classes: Producer, Work Product and Work Unit [41]. 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[7]. 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 [35]. 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).

[7] 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.

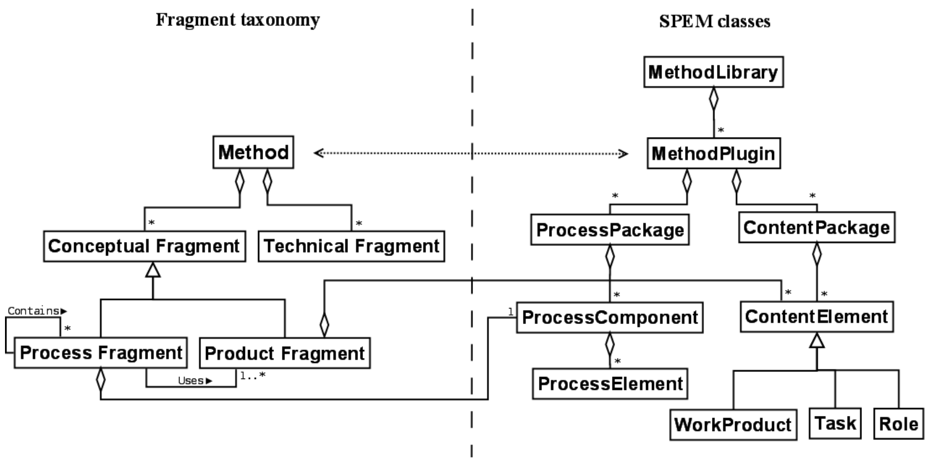


Fig. 3.2. Relationship between method fragments and SPEM classes

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 [35]) 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 [68] 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 [4], "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 [92], 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 [69]: (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 [79] 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).

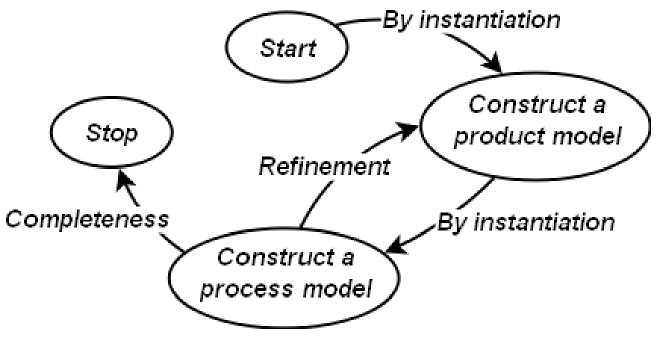


Fig. 3.3. Paradigm-based approach (adapted from [69])

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.

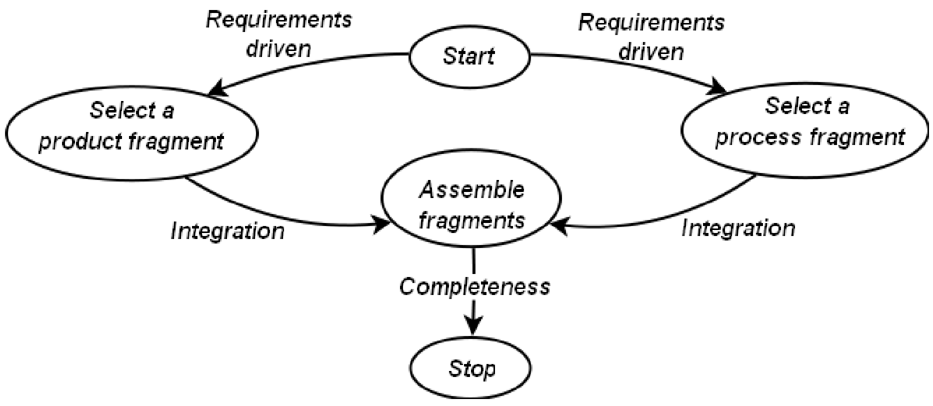


Fig. 3.4. Assembly-based approach (adapted from [69])

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[8], 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 *content 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*.

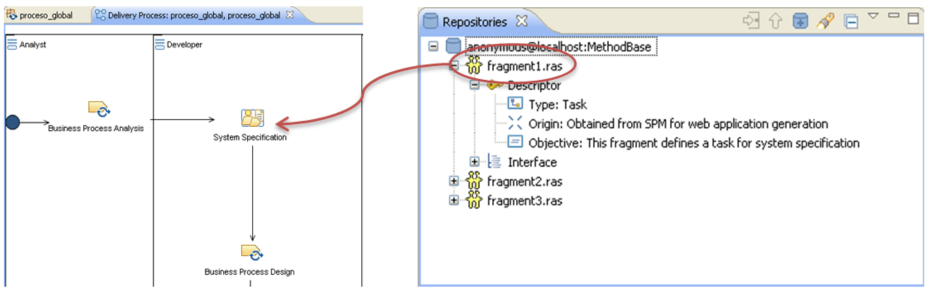

proceso_global
Delivery Process: proceso_global, proceso_global
Analyst
Developer
Business Process Analysis
System Specification
Business Process Design
Repositories
anonymous@localhost:MethodBase
fragment1.ras
Descriptor
Type: Task
Origin: Obtained from SPM for web application generation
Objective: This fragment defines a task for system specification
Interface
fragment2.ras
fragment3.ras

Fig. 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 [18]). 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.

[8] 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.

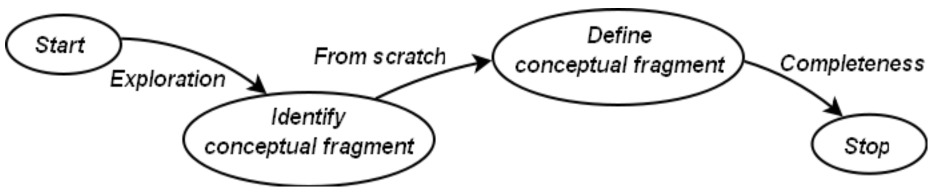


Fig. 3.6. Conceptual fragment creation (adapted from [70])

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.

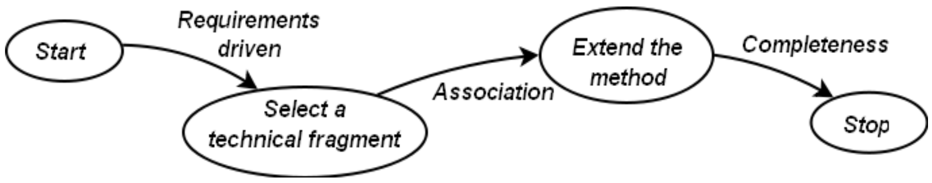


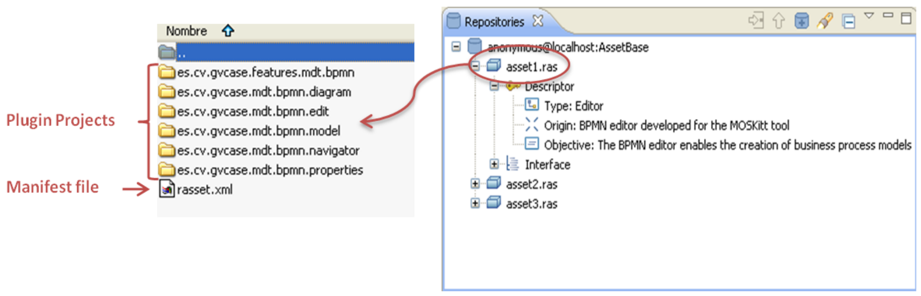
Fig. 3.7. Process model for technical fragment association

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.



Nombre
..
es.cv.gvcase.features.mdt.bpmn
es.cv.gvcase.mdt.bpmn.diagram
es.cv.gvcase.mdt.bpmn.edit
es.cv.gvcase.mdt.bpmn.model
es.cv.gvcase.mdt.bpmn.navigator
es.cv.gvcase.mdt.bpmn.properties
rasset.xml

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

Fig. 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.

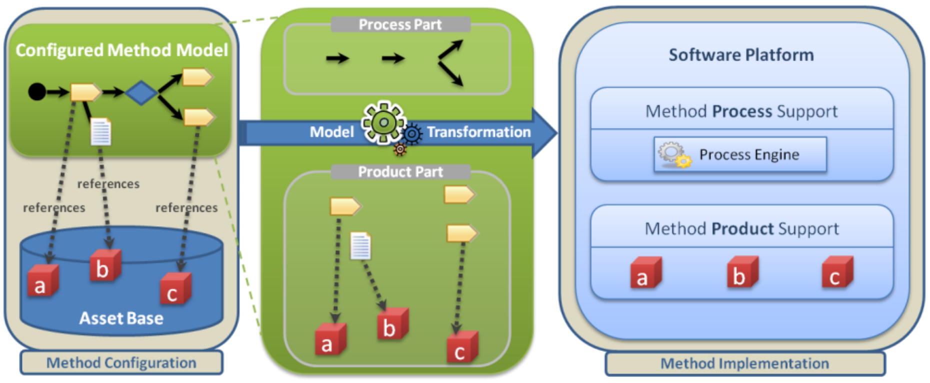


Fig. 3.9. Overview of the tool generation process

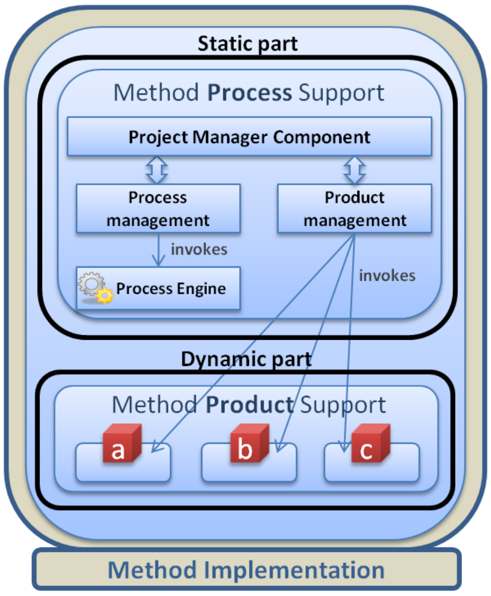


Fig. 3.10. Structure of the generated CASE tools

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[9]. 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[10].

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 [1] 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.

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

[10] 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.

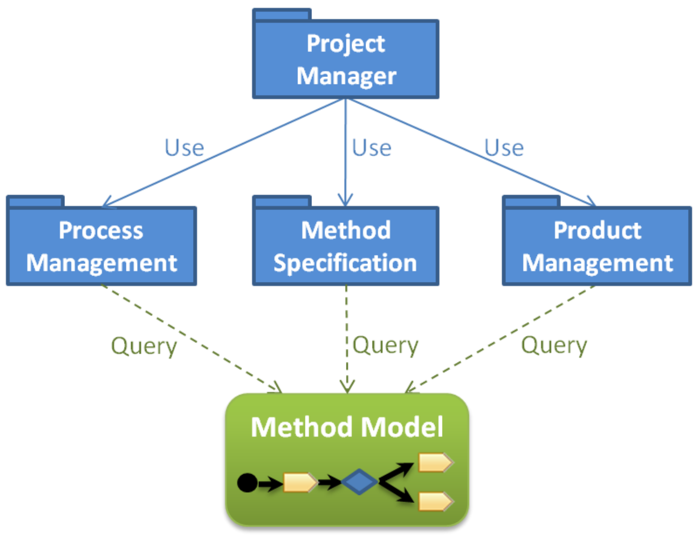


Fig. 3.11. Structure of the Project Manager Component

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.

4. A Software Architecture

CAME and metaCASE technology is still immature. Existing environments mostly represent incomplete prototypes that present important deficiencies [57]. Furthermore, these tools are generally based on rigid architectures that hinder their adaptation to new contexts of use. In order to avoid this problem, software architectures for Method Engineering supporting tools should be defined according to a set of design guidelines. In this work the following are proposed:

- **Technology-independence**: the software architecture must be defined in a technology-independent fashion in order to decouple them from technological details. This approach increases the longevity of the architecture as its components do not become obsolete on account of technology changes.

- **Modularization**: the architecture must be defined in terms of loosely-coupled components. The main benefit of this approach is that tools implementing a modular architecture are composed of separate components, and thus they are easier to extend, modify and adapt to new requirements.

- **Separation of concerns**: the software architecture must separate components that deal with Method Engineering tasks from components that deal with ISD tasks. The former components make up the structure of the CAME part, which enables tasks such as method design. On the other hand, the latter components form the CASE part, which supports ISD tasks such as system specification.

Taking these guides into account, this chapter defines a modular software architecture that identifies the set of technology-independent components (and the relationships among them) that are required to support the methodological framework presented in chapter 3. In addition, as a proof of concept of the proposal, a vertical prototype has been developed in the context of the MOSKitt platform. This prototype, called MOSKitt4ME, implements the proposed architecture and its main goal is to set the basis for the eventual development of a CAME environment that supports the design and implementation of methods, without presenting the deficiencies of current CAME and metaCASE technology.

This chapter is structured as follows: first, section 4.1 describes the requirements that the proposed architecture must address in order to provide complete support to the methodological framework. Then, section 4.2 presents the architecture in detail and also its implementation on the MOSKitt platform. Finally, section 4.3 concludes the chapter.

4.1. Architecture requirements

This section describes in detail the requirements that the proposed architecture must address in order to adequately support the methodological framework proposed in chapter 3. Specifically, this section is divided into two subsections, dealing respectively with the requirements of the CAME and CASE parts of the architecture.

4.1.1. Requirements for the CAME part

The CAME part of the architecture must include the required components to allow the method engineer to perform the method design and configuration phases of the methodological framework, and to invoke the CASE tool generation process that obtains the method implementation. Therefore, the following requirements have been identified:

Req. 1. A modeling tool for building method definitions

A modeling tool (a method editor) must be included in order to support the definition of software production methods based on a Method Engineering language such as the SPEM standard. Therefore, this tool allows the method engineer to perform the method design phase of the methodological framework.

As described in chapter 3, the method design can be performed from scratch or reusing conceptual fragments that are stored in a repository. Therefore, the modeling tool must also implement mechanisms that enable the integration of conceptual fragments into the method under construction. Furthermore, it must allow the method engineer to select parts of the method and create new conceptual fragments from these parts. This is done by means of a repository client (see req. 2).

It is also important to emphasize that the lack of a method editor is the major shortcoming of the metaCASE approach, since metaCASE tools generally focus on the method implementation. In general, metaCASE environments provide editors that enable the specification of the design notations that will be supported by the CASE tool under construction, but do not support the definition of software production methods that can be enacted in real software development projects.

Req. 2. A repository to store method fragments and mechanisms to access the repository

The method engineer must be able to reuse conceptual fragments during the method design. In addition, during the method configuration, he/she must be able to associate the tasks and products of the method with technical fragments that establish how these elements will be managed in the generated CASE tool. Therefore, mechanisms to connect the method editor and the repository containing these fragments must be provided. These mechanisms can be represented by a repository client. A repository client allows the method engineer to access the repository and search and select method fragments. For this purpose, the repository client must provide mechanisms for specifying the requirements of the fragments to retrieve. For instance, these requirements can be specified as queries that are formulated by giving values to the method fragment properties (i.e. type, origin, objective, etc.). Furthermore, the repository client must also allow the method engineer to store in the repository fragments that are created during the method design. These fragments can be later reused during the specification of other methods.

Req. 3. Mechanisms for the enactment of the Method Engineering process

The specification of software production methods is a task that must be adequately guided so that the method engineer can perform it properly. For this reason, a process that establishes the procedures and activities that must be followed during the method definition has to be defined. In order to support the execution of this process, a process engine can be included in the architecture. However, note that the inclusion of a process engine requires that the process is defined by means of an executable Process Modeling Language. Another possibility is to avoid the use of a process engine and define this process as a wizard or tutorial that textually guides the method engineer during the method definition.

Req. 4. A transformation engine

In order to automate the CASE tool generation process, a transformation engine is needed. The transformation engine is in charge of executing the model transformation that takes as input the model of the method (produced by means of the method editor) and obtains a CASE tool that supports it.

4.1.2. Requirements for the CASE part

The CASE part of the architecture must include the required components to allow the software engineer to perform the method enactment. Therefore, the following requirements have been identified:

Req. 5. Software tools that support the product part of the method

Software tools such as graphical editors, model transformations, etc. must be included in the generated CASE tool in order to support the creation and manipulation of the method products. These tools constitute the dynamic part of the CASE environments, since they depend on the method that has been specified. On the other hand, the static part corresponds to the tools that are always included in the CASE tools and, therefore, are independent of the specified method (see requirements 6 and 7).

Req. 6. Software tools that support the process part of the method

Tools such as a process engine must be included in the generated CASE tools in order to support the execution of the process part of the specified method. Thus, these tools provide a means for conducting the orchestration of the different tools that allow the creation and manipulation of the method products (see req. 5). Specifically, these tools are a static part of the generated CASE tools, in the sense that they are independent of the specified method.

It is important to note that, the method must be specified in an executable language (such as the BPMN 2.0 standard [13]) so that it can be executed in a process engine. In case the method is specified by means of a non-executable language (such as SPEM) a model transformation is required to transform the process model into an executable representation.

Req. 7. Project management mechanisms

The generated CASE tools must be endowed with a graphical user interface that allows software engineers to execute method instances (i.e. software development projects) by means of the tools that support the process part (see req. 6) and to invoke the tools that permit to create the method products (see req. 5). Like the tools that support the process part, the implementation of this graphical interface is independent of the specified method and, therefore, it is always included in the generated CASE tools.

4.2. The proposed architecture

This section describes the software architecture that is proposed in this work in order to meet the requirements presented in the previous section. Specifically, this section is divided into three subsections. First, section 4.2.1 defines the software architecture. Then, section 4.2.2 briefly presents some technological background that is needed in order to better understand how the proposed architecture has been implemented in the context of Eclipse (more specifically, on the MOSKitt platform). Finally, section 4.2.3 presents the implementation of the architecture, that is, the MOSKitt4ME prototype.

4.2.1. Conceptual definition

The proposed architecture (see Fig. 4.1) contains the set of loosely-coupled and technology-independent components that are required to support the methodological framework, i.e. to meet the requirements defined in section 4.1. These components are mainly divided into CAME components and

CASE components, and refer to the components that pertain respectively to the CAME and CASE parts of the architecture.

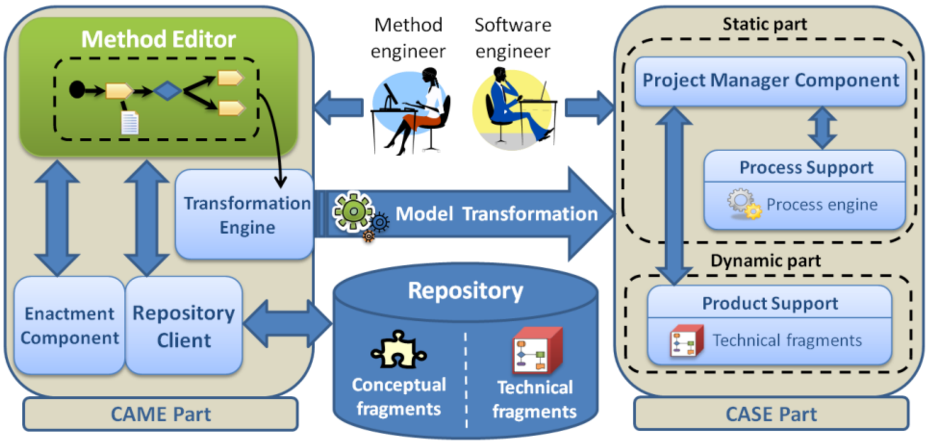


Fig. 4.1. Architecture components overview

CAME components

The CAME components make up the infrastructure of the CAME part of the architecture and are intended to meet from requirement 1 to requirement 4. Specifically, a *method editor* component (**req. 1**) has been included to allow the method engineer to perform the method design. During the construction of the method model, the method engineer can make use of the *repository* in order to reuse method fragments. For this purpose, the *repository client* (**req. 2**) is used. In general, the repository client allows the method engineer to connect to the repository, and select, reuse and store method fragments. Furthermore, the *enactment component* (**req. 3**) assists him/her during the whole method definition process. Finally, the resulting method model is fed into the *transformation engine* (**req. 4**) in order to obtain the method implementation (i.e. the CASE tool supporting the method). The method implementation is obtained by means of a model transformation that automates the generation process.

CASE components

The CASE components make up the infrastructure of the CASE part of the architecture and are intended to meet from requirement 5 to requirement 7. Specifically, the dynamic part (i.e. the components that are dependent on the specified method) is composed of the *technical fragments* (**req. 5**). These components provide support to the product part of the method. On the other hand, the static part is composed of a *process engine* (**req. 6**), which provides support to the process part of the method, and the *project manager component* (**req. 7**), which embodies the graphical user interface that allows the software engineer to perform the method enactment.

4.2.2. Technological background

This subsection provides some technological background that is needed to facilitate the understanding of the prototype that has been developed in the context of Eclipse in order to implement the proposed architecture.

The Eclipse platform

Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. Specifically, there are two features of Eclipse that turn it into a very suitable platform to support Method Engineering approaches in the field of MDD:

- The Eclipse plugin-based architecture. Everything in Eclipse is a plugin but its runtime kernel. This means that Eclipse employs plugins to provide all of its functionality. This architecture allows developers to easily build Eclipse-based applications upon the Rich Client Platform (RCP)[11]. The RCP is, roughly speaking, the minimal set of plugins required to build an Eclipse application. This approach facilitates the development of the prototype, since the different components of the architecture can be developed as separate plugins that are easy to integrate into the same platform.

- The modeling technologies and tools. Within the Eclipse community a wide range of projects aim at providing as Eclipse plugins new tools and technologies for the support of different tasks. Specifically, one of these projects is the Eclipse Modeling Project [17] which focuses on model-based development technologies. This project contributes to

[11] Rich Client Platform , http://www.eclipse.org/home/categories/rcp.php

facilitate the development of the prototype, since it provides effective solutions for applying MDD techniques.

Below, the most significant Eclipse technologies that have been used in the development of the prototype are described.

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [16] is a modeling framework and code generation facility for building tools based on a structured data model. From a meta-model specification (called the "Ecore model") described in XMI, EMF provides a generator that produces a tree-based editor, together with the set of Java classes that implement the meta-model and allow the user to create models that conform to the meta-model. Therefore, EMF has been used as the underlying technology for the construction of the method models, which are stored in XMI format and conform to the SPEM Ecore model (i.e. a SPEM meta-model implementation for Eclipse).

Eclipse Process Framework Project

The Eclipse Process Framework (EPF) [18] aims to provide an extensible framework and exemplary tools for software process engineering. Specifically, one of these tools is the EPF Composer editor, which is an Eclipse-based editor that supports the construction of SPEM models in XMI format (based on EMF). Therefore, this tool has been used as the *method editor* component of the architecture.

Plug-in Development Environment

The Plug-in Development Environment (PDE) [62] provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins and Eclipse-based applications. Therefore, the functionality provided within the PDE has been used for facilitating the construction of the CASE tools that are generated from the method specifications. Specifically, the developed prototype makes use of the Product Configuration Files. These textual files contain all the required information (list of plugins, paths of images, etc.) to automatically build Eclipse applications from them. Hence, the model transformation that obtains the CASE tool support is in fact a M2T transformation that generates a product configuration file through which the final tool is obtained.

Xpand

Xpand [92] is a statically-typed template language for implementing M2T transformations. Xpand was originally developed as part of the openArchitectureWare (oAW) project[12] before it became a component under Eclipse. Specifically, it is the language that has been used for implementing the M2T transformation that obtains product configuration files from method specifications.

4.2.3. MOSKitt4ME: An Eclipse-based CAME environment

In order to evaluate the proposed architecture, a vertical prototype, called MOSKitt4ME, has been developed in the context of Eclipse, more specifically, on the MOSKitt platform [55]. In particular, this subsection details how the different components of the architecture have been implemented in MOSKitt.

Method editor

The method editor is the software component that supports the creation of method models. In particular, the methodological framework proposes the use of the SPEM standard as the Method Engineering language to carry out this task. Therefore, MOSKitt4ME must provide a method editor that enables the creation of SPEM models. For this purpose, the EPF Composer editor [18] has been integrated in MOSKitt. Fig 4.2[13] shows a snapshot of this editor.

Repository client

The repository client component must allow the method engineer (1) to connect to the repository, to (2) search and select method fragments for their use during the method design and configuration phases, and (3) to store newly created fragments. For this purpose, a repository client has been implemented as an Eclipse view. This view shows in a tree-based fashion the content of the repository it is connected to and provides searching capabilities based on fragment properties. In order to illustrate this idea, Fig. 4.3 and Fig. 4.4 show

[12] http://www.openarchitectureware.org/

[13] Also available at https://users.dsic.upv.es/~vtorres/moskitt4me/

this Eclipse view connected to the Method Base and Asset Base repositories respectively.

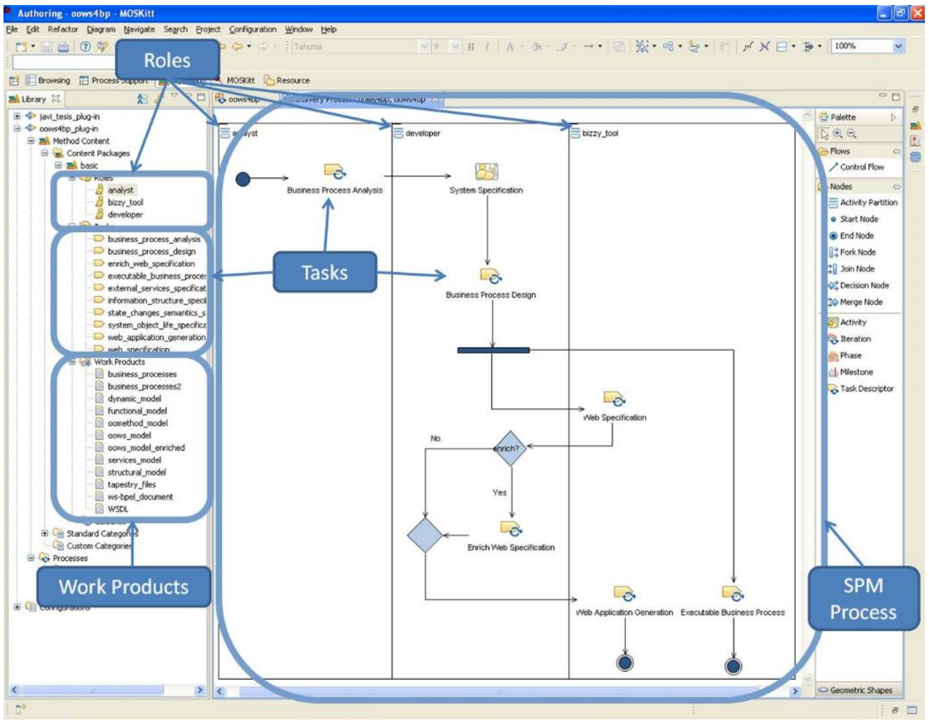
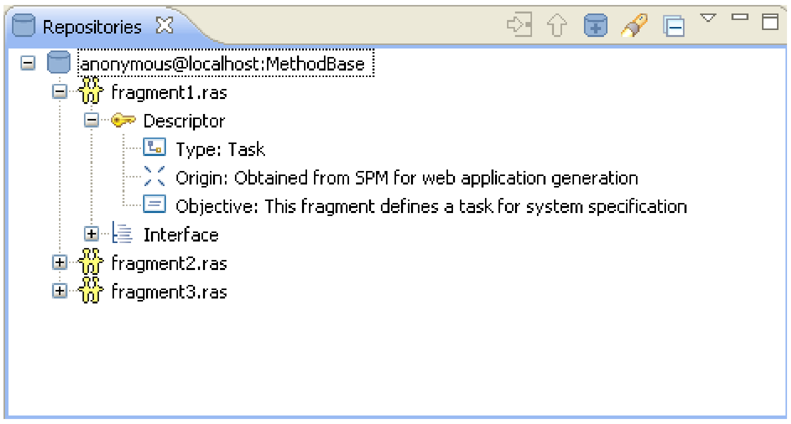


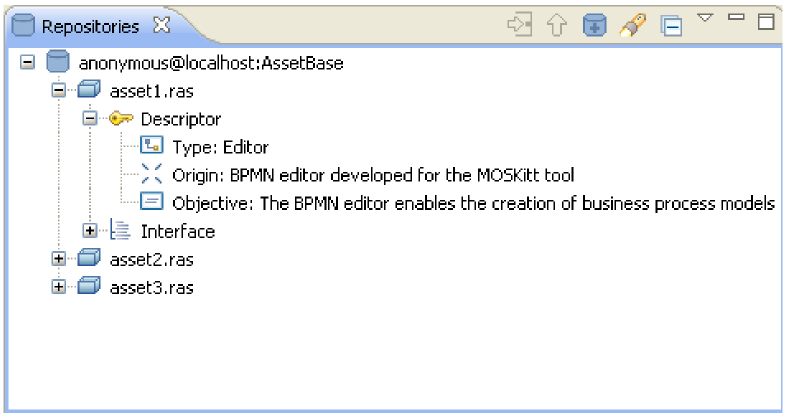
Fig. 4.2. EPF Composer editor in MOSKitt



Repositories

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

Fig. 4.3. Repository client (Method Base)



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

Fig. 4.4. Repository client (Asset Base)

Enactment component

A process engine has not been integrated into the prototype to guide method engineers during the method definition. Instead, two eclipse cheatsheets have been defined to assist during the method design and configuration phases of the methodological framework.

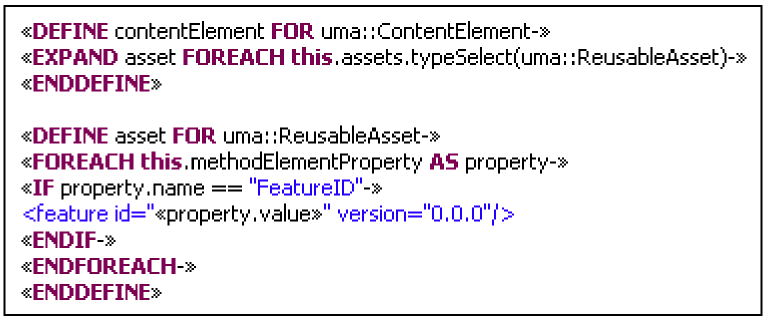
Transformation engine

In order to support the execution of the model transformation that generates the CASE tool support from method models, Xpand has been installed in the prototype. The Xpand plugins implement, among other things, the transformation engine that supports the execution of Xpand transformations.

Specifically, the model transformation has been implemented in the prototype as a M2T transformation that takes as input a SPEM model and obtains a product configuration file through which a MOSKitt reconfiguration supporting the method is obtained. As an example, two Xpand rules of the transformation are shown in Fig. 4.5. In these rules the list of features[14] of the product configuration file is generated. The first rule is invoked for each instance of the SPEM 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

[14] A feature is a group of Eclipse plugins

during the technical fragment association and contains the identifier of the feature packaged in the fragment.



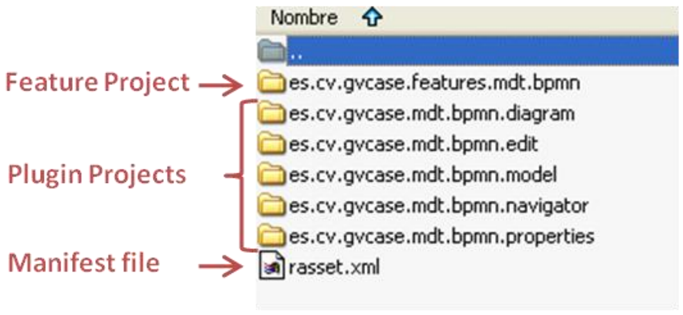
```
«DEFINE contentElement FOR uma::ContentElement-»
«EXPAND asset FOREACH this.assets.typeSelect(uma::ReusableAsset)-»
«ENDDEFINE»

«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»
```

Fig. 4.5. Excerpt of the M2T transformation

Technical fragments

Technical fragments are editors, transformations, etc. that provide support to the product part of the method in the generated CASE tools. These fragments are stored in the Asset Base repository as reusable assets that contain the Eclipse plugins that implement the encapsulated tool and the feature that groups these plugins (see Fig. 4.6). In order to install these plugins in the CASE tools, the M2T transformation must include in the product configuration file the features encapsulated in the fragments. This is done in the rules shown in Fig. 4.5.



```
..
es.cv.gvcase.features.mdt.bpmn
es.cv.gvcase.mdt.bpmn.diagram
es.cv.gvcase.mdt.bpmn.edit
es.cv.gvcase.mdt.bpmn.model
es.cv.gvcase.mdt.bpmn.navigator
es.cv.gvcase.mdt.bpmn.properties
rasset.xml
```

Fig. 4.6. Technical fragment

Process Engine

The process engine is the component in charge of the execution of method instances, that is, it gives support to the process part of the method in the generated CASE tools. Up to now, the process engine has been implemented in MOSKitt4ME as a light-weight process engine that keeps the state of the running method instances. As future work, the integration of the Activiti engine [1] into MOSKitt4ME is being planned. The use of Activiti will require the definition of a model transformation to map SPEM models into BPMN 2.0 models that can be executed by the engine.

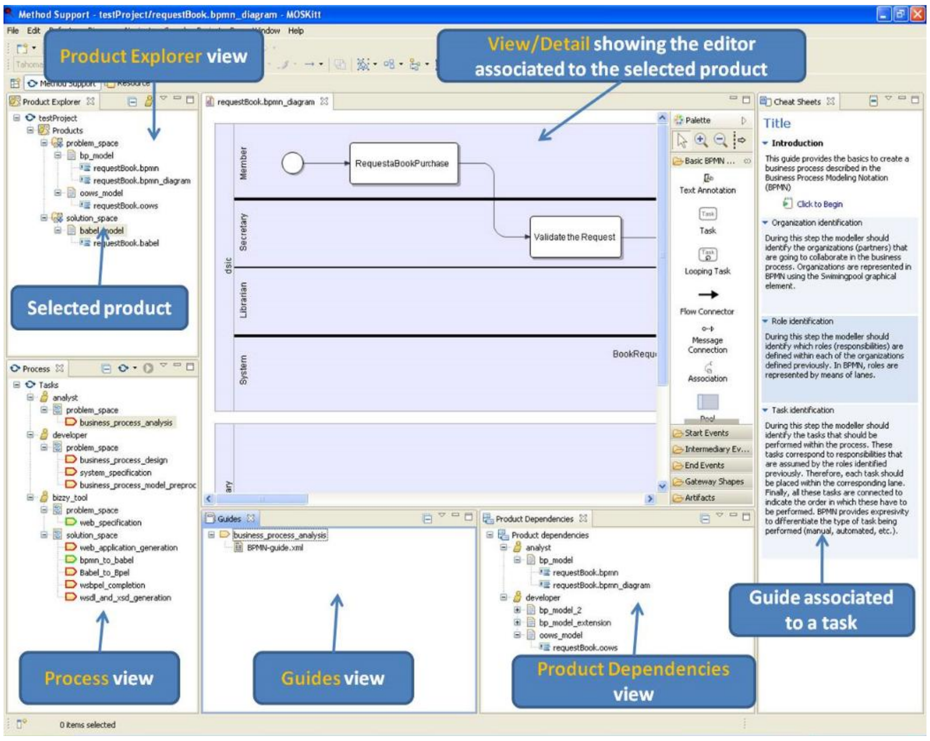
Project Manager Component

The Project Manager Component endows the generated CASE tools with a graphical user interface composed of a set of Eclipse views (see Fig. 4.7[15]). 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 technical fragment) or manually by the software engineer (by means of the software tool 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 technical fragments that were associated to tasks during the method configuration phase.

[15] Also available at https://users.dsic.upv.es/~vtorres/moskitt4me/

- *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.



Title

Introduction

This guide provides the basics to create a business process described in the Business Process Modeling Notation (BPMN)

Organization identification

During this step the modeller should identify the organizations (partners) that are going to collaborate in the business process. Organizations are represented in BPMN using the Swimingpool graphical element.

Role identification

During this step the modeller should identify which roles (responsibilities) are defined within each of the organizations defined previously. In BPMN, roles are represented by means of lanes.

Task identification

During this step the modeller should identify the tasks that should be performed within the process. These tasks correspond to responsibilities that are assumed by the roles identified previously. Therefore, each task should be placed within the corresponding lane. Finally, all these tasks are connected to indicate the order in which these have to be performed. BPMN provides expresivity to differentiate the type of task being performed (manual, automated, etc.).

Fig. 4.7. Project Manager Component

4.3. Conclusions

Developing software systems is a highly complex endeavor and CAME and metaCASE environments are no exception. A solution that properly handles this complexity is software architecting. One of the main benefits of a software architecture is that it provides an abstraction of the system that establish how it must be structured and, thus, allow developers to focus only on those elements that are significant. Therefore, in order to reduce the complexity that entails the development of tools that support Method Engineering, this chapter proposes a software architecture that establishes the series of components that are required to support the methodological framework presented in chapter 3.

Furthermore, a vertical prototype called MOSKitt4ME has been developed in the context of the MOSKitt platform as an implementation of the architecture. The development of this prototype has a threefold benefit. First, it helps to evaluate the proposed architecture. Secondly, it sets the basis for the eventual development of a complete CAME environment. Finally, stakeholders within the MOSKitt community can use the prototype and provide feedback that can be used for the refinement of the architecture and the methodological framework.

5. A Case Study

This chapter presents a case study that has been developed to validate the methodological framework and the software architecture proposed in this thesis. In this case study MOSKitt4ME has been used for specifying a software production method and generating its supporting CASE environment. In particular, the software production method comes from [90]. This method defines an MDD approach for the generation of web applications supporting business process specifications.

This chapter is structured as follows: first, section 5.1 provides an overview of the case study. Then, section 5.2 describes in detail how it has been developed in MOSKitt4ME. Finally, section 5.3 outlines some conclusions.

5.1. The OOWS-BP method

OOWS-BP [90] is a software production method that results from extending the OOWS web engineering approach [23]. This extension introduces and modifies some of the existing steps in order to deal with the execution of business processes. Thus, OOWS-BP embodies an MDD approach for the generation of business process-driven web applications from conceptual models. Briefly presented, the OOWS-BP method (see Fig. 5.1) involves the participation of three different roles: the *analyst* and the *developer* (related to human beings), and the *bizzy tool* (which represents the software system). The process is started by the analyst who specifies, by means of the BPMN notation, the business processes that have to be supported in the web application. This specification constitutes a non-executable version of the process models, which require more details to be deployed and run in a process engine. Then, in the next step, the developer performs the system specification, i.e. the business process is defined in terms of the OO-Method models [60] and the OOWS services model. Once the system specification is finished, these models are used by model-to-model (M2M) transformations that generate the OOWS navigational and presentation models, and the business process in WS-BPEL (an executable representation of the business process). Finally, the Tapestry[16] files that implement the web application are obtained from the OOWS models (which can be manually modified by the developer).

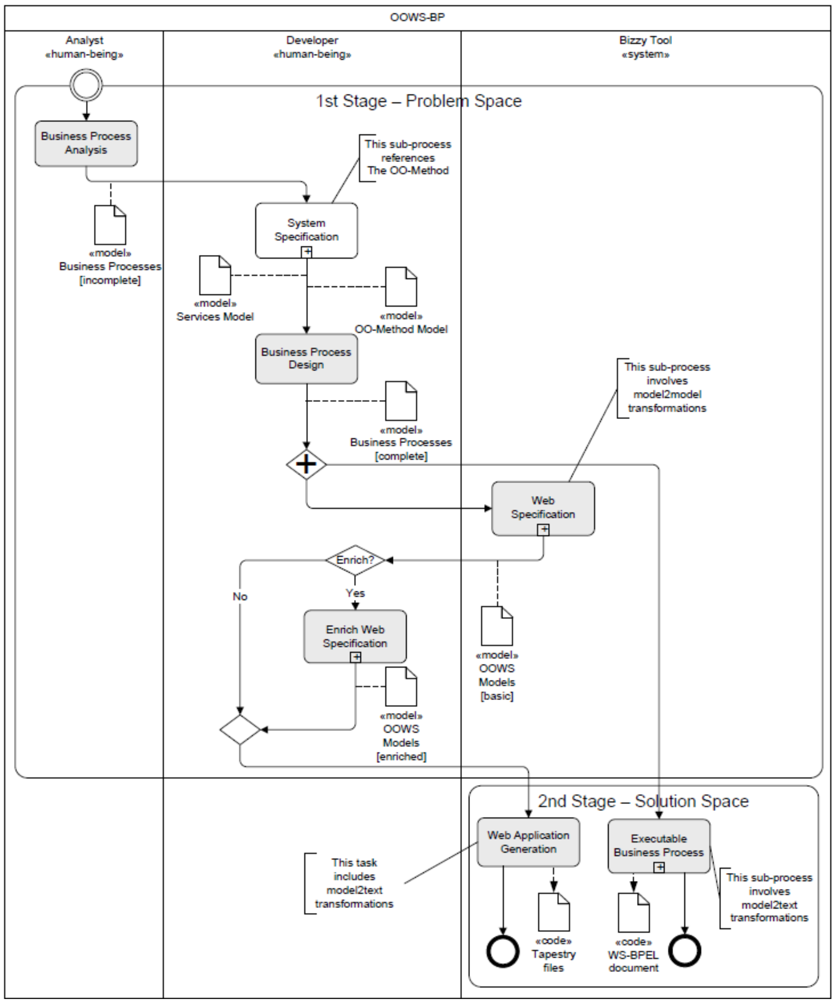

OOWS-BP
Analyst «human-being»
Developer «human-being»
Bizzy Tool «system»
1st Stage – Problem Space
Business Process Analysis
This sub-process references The OO-Method
System Specification
«model» Business Processes [incomplete]
«model» Services Model
«model» OO-Method Model
Business Process Design
This sub-process involves model2model transformations
«model» Business Processes [complete]
Web Specification
Enrich?
No
Yes
Enrich Web Specification
«model» OOWS Models [basic]
«model» OOWS Models [enriched]
2nd Stage – Solution Space
Web Application Generation
Executable Business Process
This task includes model2text transformations
This sub-process involves model2text transformations
«code» Tapestry files
«code» WS-BPEL document

Fig. 5.1. The OOWS-BP method

[16] Tapestry, http://tapestry.apache.org/

5.2. Development of the case study

This section details how the case study has been developed in MOSKitt4ME following the methodological framework presented in chapter 3. Specifically, the section is divided into three subsections (based on the framework phases) in order to describe how the model of the method is built and how the supporting CASE tool is obtained from this model.

5.2.1. Method design

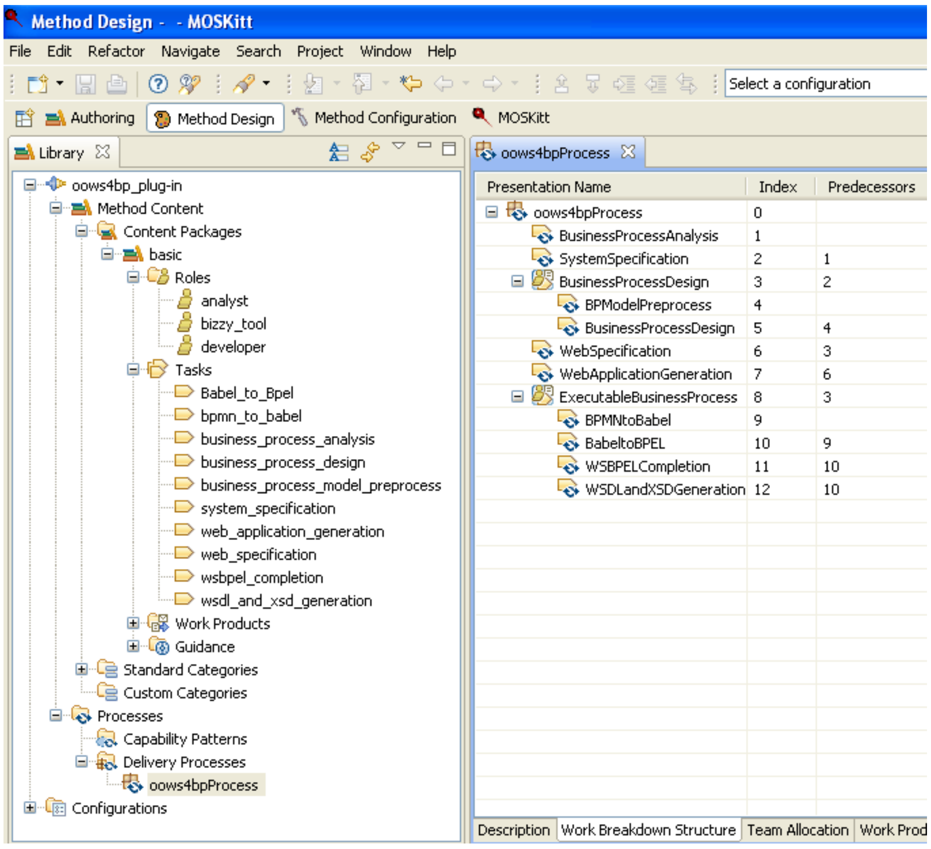
In this phase, the EPF Composer editor is used for the creation of the method model. Following the process defined in chapter 3 (paradigm-based approach), the method model is created in two steps, (1) the definition of the product model and (2) the definition of the process model.

In this proposal, the product model and the method content part of a SPEM method are considered analogous, therefore, the first step has been to create by means of the EPF Composer the method content of the OOWS-BP method. Since at this stage the method model is specified without detailing the techniques, languages and notations that will be used during the method enactment, this part of the model is composed of generic products (e.g. *business process model*, *services model*, etc.), tasks (e.g. *business process analysis*, *system specification*, etc.) and roles (e.g. *analyst*, *developer*, etc.).

Once the method content is defined, the process model is built. The process model corresponds to the method process part of a SPEM method. Therefore, the second step has been to create by means of the EPF Composer the Work Breakdown Structure that establishes the tasks execution order.

Furthermore, as described in chapter 3, during the construction of the method model it is possible to reuse conceptual fragments stored in the Method Base repository (assembly-based approach). Specifically, during the definition of the OOWS-BP method model a product fragment containing the task *system specification* has been used. In order to do so, first it has been extracted from the repository by means of the repository client, and then, its content (i.e. the method task) has been automatically integrated into the method content part of the model.

Fig. 5.2 shows a snapshot of the EPF Composer containing the OOWS-BP method model resulting from the method design phase. On the left part of the figure, the *Library* view shows some method content elements (i.e. tasks, roles, etc.) in a tree viewer. On the right part, details of the process are depicted as a Work Breakdown Structure.

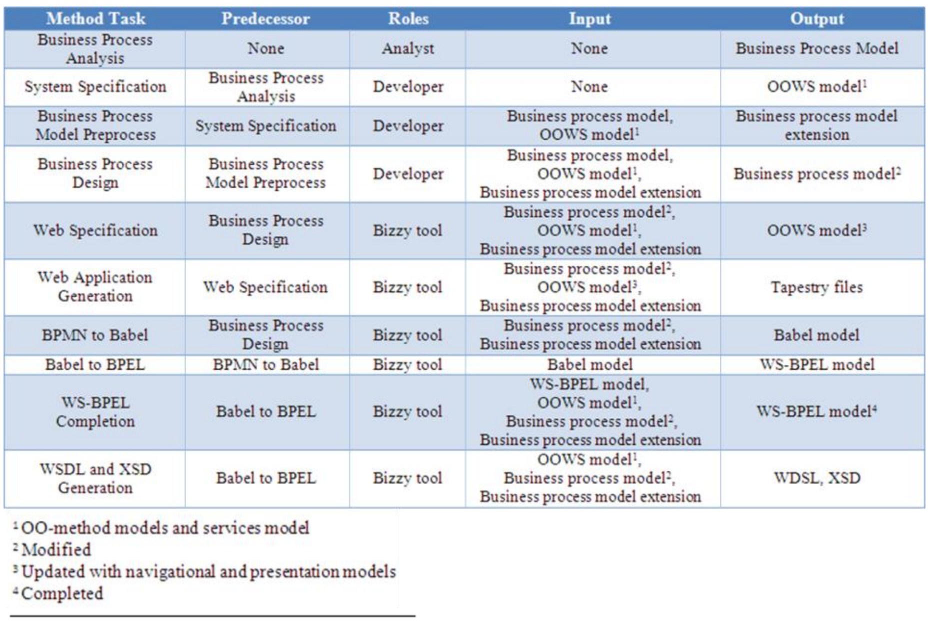


- oows4bp_plug-in
  - Method Content
    - Content Packages
      - basic
        - Roles
          - analyst
          - bizzy_tool
          - developer
        - Tasks
          - Babel_to_Bpel
          - bpmn_to_babel
          - business_process_analysis
          - business_process_design
          - business_process_model_preprocess
          - system_specification
          - web_application_generation
          - web_specification
          - wsbpel_completion
          - wsdl_and_xsd_generation
        - Work Products
        - Guidance
    - Standard Categories
    - Custom Categories
  - Processes
    - Capability Patterns
    - Delivery Processes
      - oows4bpProcess
- Configurations

Presentation Name	Index	Predecessors
oows4bpProcess	0	
BusinessProcessAnalysis	1	
SystemSpecification	2	1
BusinessProcessDesign	3	2
BPModelPreprocess	4	
BusinessProcessDesign	5	4
WebSpecification	6	3
WebApplicationGeneration	7	6
ExecutableBusinessProcess	8	3
BPMNtoBabel	9	
BabeltoBPEL	10	9
WSBPELCompletion	11	10
WSDLandXSDGeneration	12	10

Fig. 5.2. Case study specification in the EPF Composer

Moreover, table 5.1 provides further details about all these tasks. Specifically, this table contains the tasks predecessors, the performing roles and the input/output products. In addition, all the tasks are briefly described below.



Method Task	Predecessor	Roles	Input	Output
Business Process Analysis	None	Analyst	None	Business Process Model
System Specification	Business Process Analysis	Developer	None	OOWS model[1]
Business Process Model Preprocess	System Specification	Developer	Business process model, OOWS model[1]	Business process model extension
Business Process Design	Business Process Model Preprocess	Developer	Business process model, OOWS model[1], Business process model extension	Business process model[2]
Web Specification	Business Process Design	Bizzy tool	Business process model[2], OOWS model[1], Business process model extension	OOWS model[3]
Web Application Generation	Web Specification	Bizzy tool	Business process model[2], OOWS model[3], Business process model extension	Tapestry files
BPMN to Babel	Business Process Design	Bizzy tool	Business process model[2], Business process model extension	Babel model
Babel to BPEL	BPMN to Babel	Bizzy tool	Babel model	WS-BPEL model
WS-BPEL Completion	Babel to BPEL	Bizzy tool	WS-BPEL model, OOWS model[1], Business process model[2], Business process model extension	WS-BPEL model[4]
WSDL and XSD Generation	Babel to BPEL	Bizzy tool	OOWS model[1], Business process model[2], Business process model extension	WDSL, XSD

[1] OO-method models and services model
[2] Modified
[3] Updated with navigational and presentation models
[4] Completed

Table 5.1. OOWS-BP tasks

Business Process Analysis

The analyst specifies as a non-executable process model the business process that will be supported by the generated web application.

System Specification

The developer defines the business process in terms of the OO-Method models and the services model.

Business Process Design (subprocess)

The developer completes the business process model with additional information.

Business Process Model Preprocess

The developer builds an extension of the business process model in order to specify additional information that is not supported by the notation used to create the business process model.

Business Process Design

The developer completes the business process model with information that was not specified by the analyst.

Web Specification

This task automatically generates from the previously built models a navigational model and a presentation model, that is, the specification of the web application as defined by the OOWS approach [23].

Web Application Generation

This task automatically generates the web application from its specification. This application is implemented by means of the framework Tapestry.

Executable Business Process (subprocess)

This task embodies a transformation chain that obtains the executable WS-BPEL specification from the business process model.

BPMN to Babel

This automatic task executes a M2M transformation that obtains an intermediate representation of the business process model (babel notation).

Babel to BPEL

This automatic task executes a M2M transformation that transforms the business process model (in babel notation) into an executable WS_BPEL model.

WS-BPEL Completion

This task executes a M2M transformation that completes the WS-BPEL model so that it can be imported by the process engine.

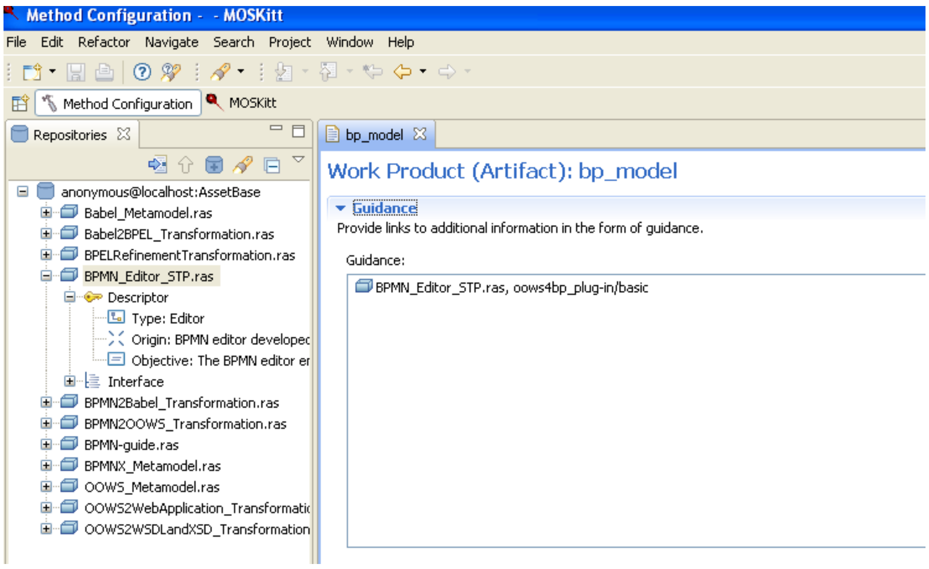
WSDL and XSD Generation

This task generates the WSDL and XSD files that complete the WS-BPEL model in order to make it deployable. Specifically, the WSDL files define the interface associated to the new service defined by the WS-BPEL and the XSD files define the data types used by it.

5.2.2. Method configuration

Once the product and process parts of the method model have been specified, the method configuration phase can start. Following the process defined in chapter 3 for method configuration, in this phase the method engineer must make use of the repository client in order to (1) select technical fragments and (2) associate them with tasks and products of the method model. This association represents that the technical fragments must be included in the generated CASE tool in order to provide support to the tasks and products they are associated to.

Fig. 5.3 shows the Eclipse view that implements the repository client in MOSKitt4ME. This view is showing the fragments that support the OOWS-BP tasks and products. Moreover, the right part of Fig. 5.3 shows an example of association of a technical fragment (BPMN editor) with a method product (business process model).


Method Configuration - - MOSKitt
File Edit Refactor Navigate Search Project Window Help
Method Configuration
MOSKitt
Repositories
bp_model

anonymous@localhost:AssetBase
Babel_Metamodel.ras
Babel2BPEL_Transformation.ras
BPELRefinementTransformation.ras
BPMN_Editor_STP.ras
Descriptor
Type: Editor
Origin: BPMN editor developec
Objective: The BPMN editor er
Interface
BPMN2Babel_Transformation.ras
BPMN2OOWS_Transformation.ras
BPMN-guide.ras
BPMNX_Metamodel.ras
OOWS_Metamodel.ras
OOWS2WebApplication_Transformatic
OOWS2WSDLandXSD_Transformation

Work Product (Artifact): bp_model

Guidance

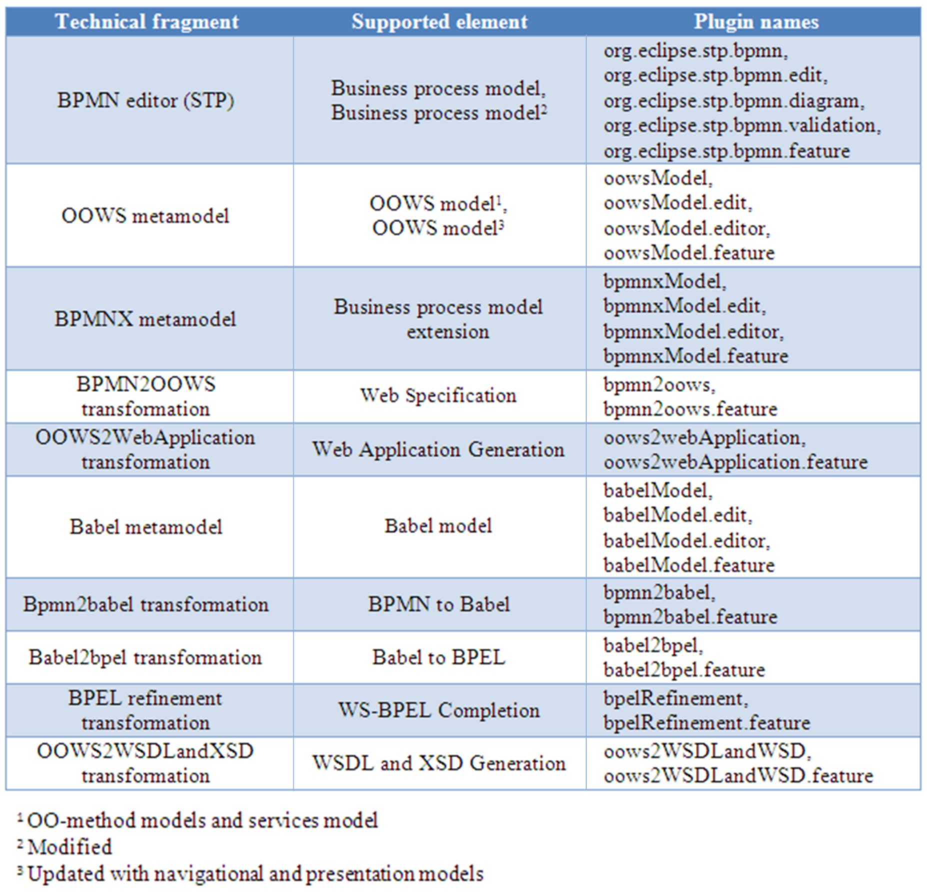
Provide links to additional information in the form of guidance.

Guidance:

BPMN_Editor_STP.ras, oows4bp_plug-in/basic

Fig. 5.3. Technical fragments supporting the case study

In order to give more information about all these fragments, table 5.2 shows for each of them the supported method elements and the Eclipse plugins it contains. In addition, a brief description of the fragments is given below. Finally, the overall associations between the OOWS-BP elements and the technical fragments are summarized in tables 5.3 and 5.4.



Technical fragment	Supported element	Plugin names
BPMN editor (STP)	Business process model, Business process model[2]	org.eclipse.stp.bpmn, org.eclipse.stp.bpmn.edit, org.eclipse.stp.bpmn.diagram, org.eclipse.stp.bpmn.validation, org.eclipse.stp.bpmn.feature
OOWS metamodel	OOWS model[1], OOWS model[3]	oowsModel, oowsModel.edit, oowsModel.editor, oowsModel.feature
BPMNX metamodel	Business process model extension	bpmnxModel, bpmnxModel.edit, bpmnxModel.editor, bpmnxModel.feature
BPMN2OOWS transformation	Web Specification	bpmn2oows, bpmn2oows.feature
OOWS2WebApplication transformation	Web Application Generation	oows2webApplication, oows2webApplication.feature
Babel metamodel	Babel model	babelModel, babelModel.edit, babelModel.editor, babelModel.feature
Bpmn2babel transformation	BPMN to Babel	bpmn2babel, bpmn2babel.feature
Babel2bpel transformation	Babel to BPEL	babel2bpel, babel2bpel.feature
BPEL refinement transformation	WS-BPEL Completion	bpelRefinement, bpelRefinement.feature
OOWS2WSDLandXSD transformation	WSDL and XSD Generation	oows2WSDLandWSD, oows2WSDLandWSD.feature

[1] OO-method models and services model
[2] Modified
[3] Updated with navigational and presentation models

Table 5.2. OOWS-BP technical fragments

BPMN editor (STP)

This technical fragment contains the Eclipse plugins that implement the BPMN graphical editor developed as part of the SOA Tools Platform Project (STP) [86].

OOWS metamodel

This fragment contains the plugins that implement the OOWS metamodel.

BPMNX metamodel

This fragment contains the plugins that implement the extension of the BPMN metamodel.

BPMN2OOWS transformation

This technical fragment encapsulates the M2M transformation implemented in ATL [5] that obtains the OOWS navigational and presentation models.

OOWS2WebApplication transformation

This fragment encapsulates the M2T transformation implemented in MOFScript [54] that obtains the final web application from the conceptual models (OOWS, BPMN, etc.).

Babel metamodel

This fragment contains the plugins that implement the Babel metamodel. This metamodel enables the creation of BPMN models that can be transformed in WS-BPEL models by the transformation bpmn2bpel.

Bpmn2babel transformation

This fragment contains the plugins that implement the ATL M2M transformation that automatically obtains a babel model from a business process model specified in BPMN.

Babel2bpel transformation

This fragment contains the M2M transformation implemented in ATL that obtains the WS-BPEL model from the Babel model.

BPEL refinement transformation

This fragment encapsulates the plugins that implement the ATL M2M transformation that completes the WS-BPEL model so that it can be imported in the process engine.

OOWS2WSDLandXSD transformation

This fragment contains the M2T transformation implemented in MOFScript that obtains the WSDL and XSD files associated to the WS-BPEL process.

Summary

In order to provide an overview of the products and task of the method and which technical fragments provide support to them, table 5.3 shows the associations between products and technical fragments and table 5.4 the associations between tasks and technical fragments.

Method Product	Technical Fragment
Business process model	BPMN editor (STP)
OOWS model[1]	OOWS metamodel
Business process model extension	BPMNX metamodel
Business process model[2]	BPMN editor (STP)
OOWS model[3]	OOWS metamodel
Tapestry files	No fragment has been specified
Babel model	Babel metamodel
WS-BPEL model	No fragment has been specified
WS-BPEL model[4]	No fragment has been specified
WDSL and XSD	No fragment has been specified

[1] OO-method models and services model
[2] Modified
[3] Updated with navigational and presentation models
[4] Completed

Table 5.3. Relationship between products and technical fragments

Method Task	Technical Fragment
Business Process Analysis	Guide (optional)
System Specification	Guide (optional)
Business Process Model Preprocess	Guide (optional)
Business Process Design	Guide (optional)
Web Specification	BPMN2OOWS transformation
Web Application Generation	OOWS2WebApplication transformation
BPMN to Babel	Bpmn2babel transformation
Babel to BPEL	Babel2bpel transformation
WS-BPEL Completion	BPEL refinement transformation
WSDL and XSD Generation	OOWS2WSDLandXSD transformation

Table 5.4. Relationship between tasks and technical fragments

5.2.3. Method implementation

In this phase, the method engineer invokes the model transformation that obtains from the configured method model the CASE tool that supports the method. As described in chapter 4, this transformation has been implemented in MOSKitt4ME as a M2T transformation that obtains from the method model a product configuration file through which the final tool is obtained. This tool is a MOSKitt reconfiguration that only contains the required plugins to support the method (i.e. the plugins contained in the technical fragments, the process engine and the Eclipse views that compose the GUI). Specifically, the M2T transformation is invoked by means of the MOSKitt transformation manager, which is shown in Fig. 5.4. Through this Eclipse view, all the transformations registered in MOSKitt can be launched.

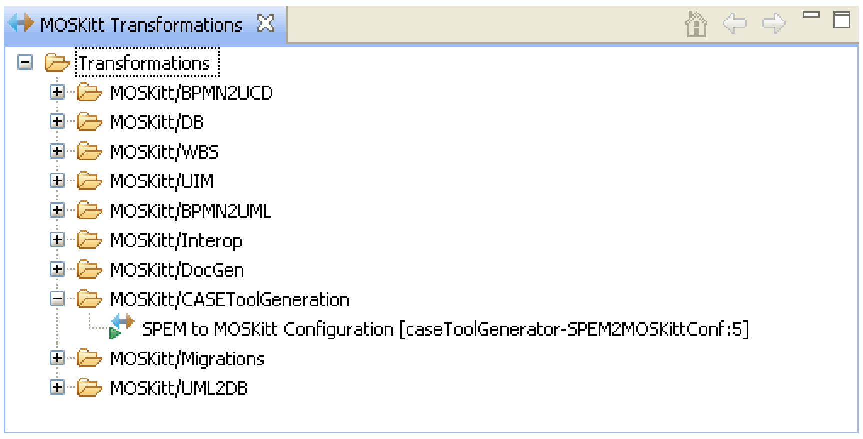
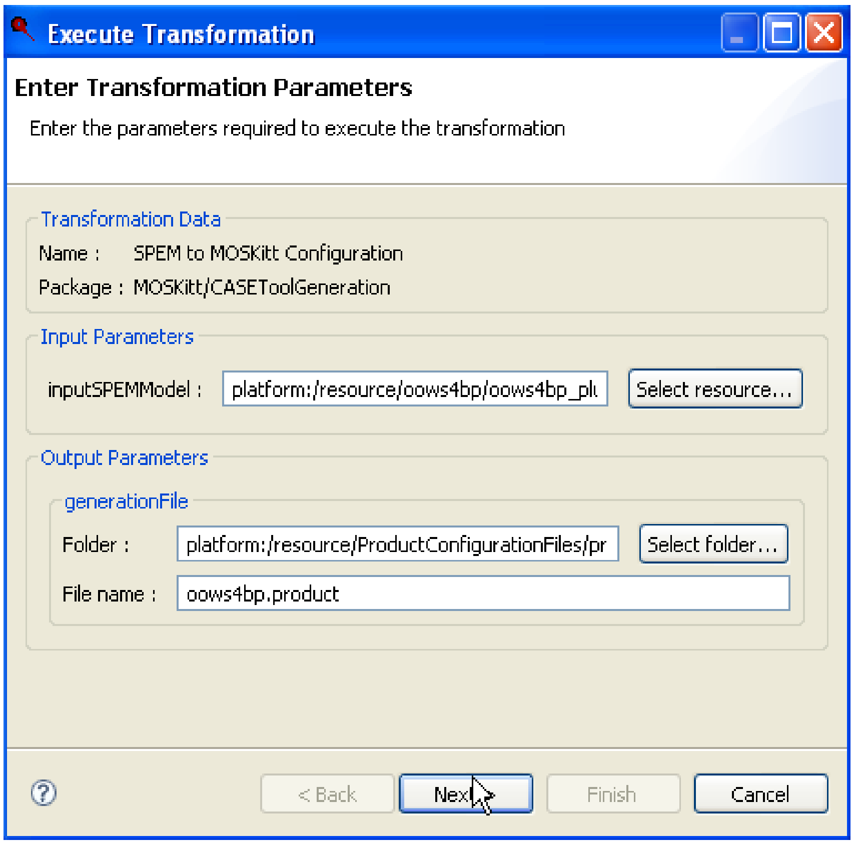

MOSKitt Transformations
Transformations
MOSKitt/BPMN2UCD
MOSKitt/DB
MOSKitt/WBS
MOSKitt/UIM
MOSKitt/BPMN2UML
MOSKitt/Interop
MOSKitt/DocGen
MOSKitt/CASEToolGeneration
SPEM to MOSKitt Configuration [caseToolGenerator-SPEM2MOSKittConf:5]
MOSKitt/Migrations
MOSKitt/UML2DB

Fig. 5.4. MOSKitt transformation manager

When selecting the *SPEM2MOSKittConf* transformation, a wizard is opened (see Fig. 5.5). Specifically, in this wizard the input and output parameters of the transformation can be specified. The input parameter corresponds to the SPEM model resulting from the method configuration phase. The output parameter corresponds to the product configuration file through which the MOSKitt reconfiguration supporting the method will be obtained.


Execute Transformation

Enter Transformation Parameters

Enter the parameters required to execute the transformation

Transformation Data

Name : SPEM to MOSKitt Configuration

Package : MOSKitt/CASEToolGeneration

Input Parameters

inputSPEMModel : platform:/resource/oows4bp/oows4bp_pl

Output Parameters

generationFile

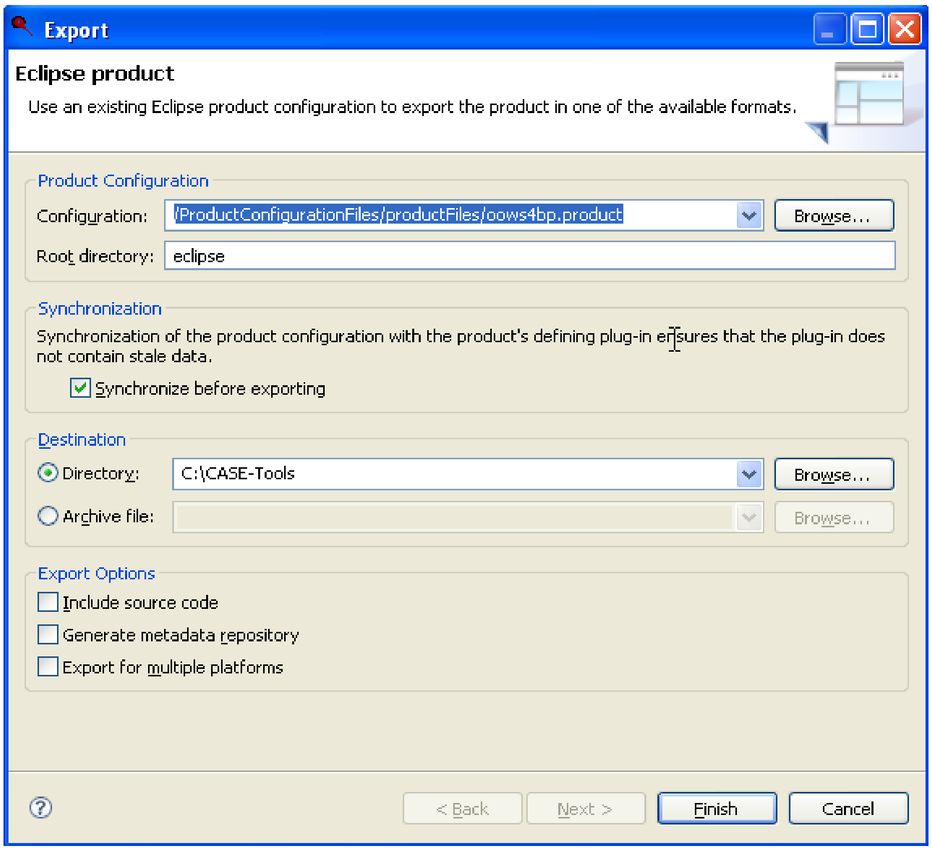
Folder : platform:/resource/ProductConfigurationFiles/pr

File name : oows4bp.product

Fig. 5.5. MOSKitt transformation wizard

Once the product configuration file is generated, the export wizard is automatically launched. This wizard is shown in Fig. 5.6. Specifically, it allows the method engineer to generate the final CASE tool from the product configuration file. For this purpose, at least the following information must be specified:

- **Configuration**: The product configuration file. It is automatically set when the wizard is opened.
- **Root directory**: Name of the folder hosting the generated tool. By default this folder is named *eclipse*.
- **Destination**: Path of the file system where the folder *Root directory* will be placed. If selected the option “Archive file” a package (zip file) of the tool is obtained.


Export

Eclipse product

Use an existing Eclipse product configuration to export the product in one of the available formats.

Product Configuration

Configuration: /ProductConfigurationFiles/productFiles/oows4bp.product

Root directory: eclipse

Synchronization

Synchronization of the product configuration with the product's defining plug-in ensures that the plug-in does not contain stale data.

[✓] Synchronize before exporting

Destination

(●) Directory: C:\CASE-Tools

( ) Archive file:

Export Options

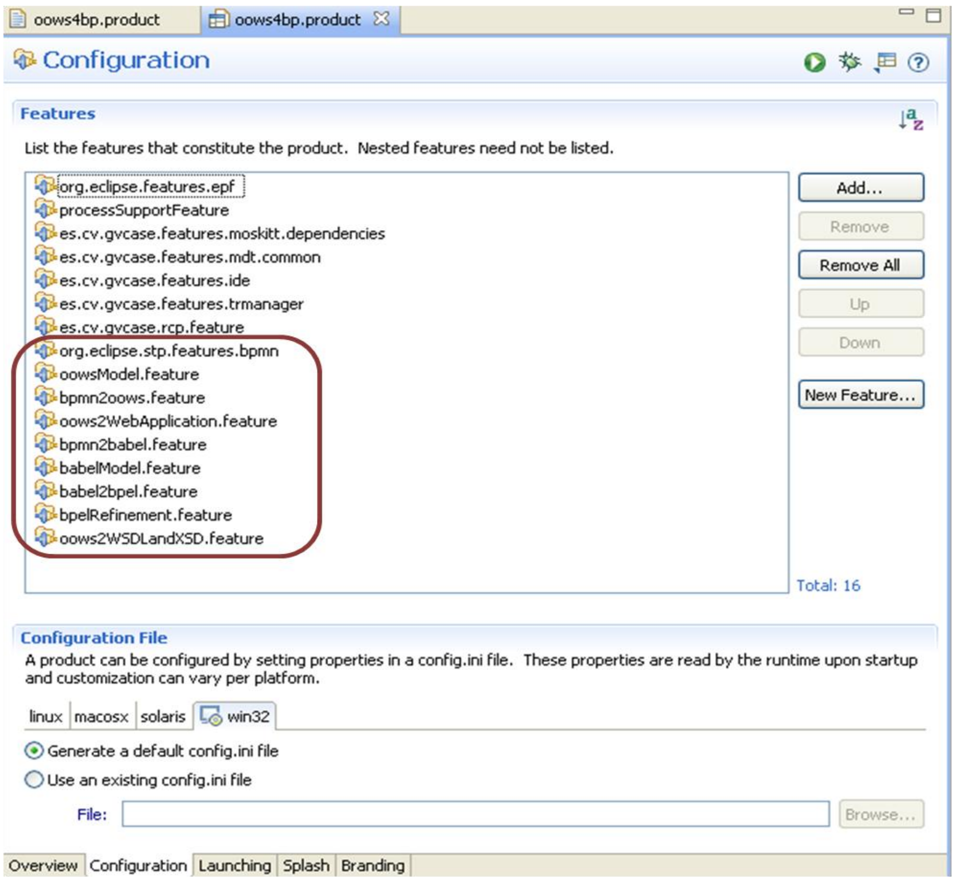
[ ] Include source code

[ ] Generate metadata repository

[ ] Export for multiple platforms

Fig. 5.6. Export wizard

In order to illustrate the contents of the product configuration file, Fig 5.7 shows the list of features that establish the plugins that must be included in the final CASE tool. Specifically, the features that correspond to the plugins contained in the technical fragments have been emphasized. The remaining features correspond to software dependencies and to the Project Manager Component.


oows4bp.product

oows4bp.product

Configuration

Features

List the features that constitute the product. Nested features need not be listed.

- org.eclipse.features.epf
- processSupportFeature
- es.cv.gvcase.features.moskitt.dependencies
- es.cv.gvcase.features.mdt.common
- es.cv.gvcase.features.ide
- es.cv.gvcase.features.trmanager
- es.cv.gvcase.rcp.feature
- org.eclipse.stp.features.bpmn
- oowsModel.feature
- bpmn2oows.feature
- oows2WebApplication.feature
- bpmn2babel.feature
- babelModel.feature
- babel2bpel.feature
- bpelRefinement.feature
- oows2WSDLandXSD.feature

Total: 16

Configuration File

A product can be configured by setting properties in a config.ini file. These properties are read by the runtime upon startup and customization can vary per platform.

linux | macosx | solaris | win32

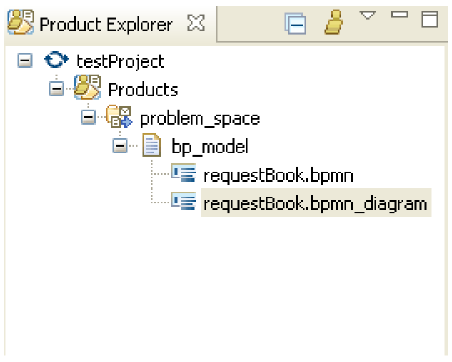
[✓ Generate a default config.ini file]

Use an existing config.ini file

File:

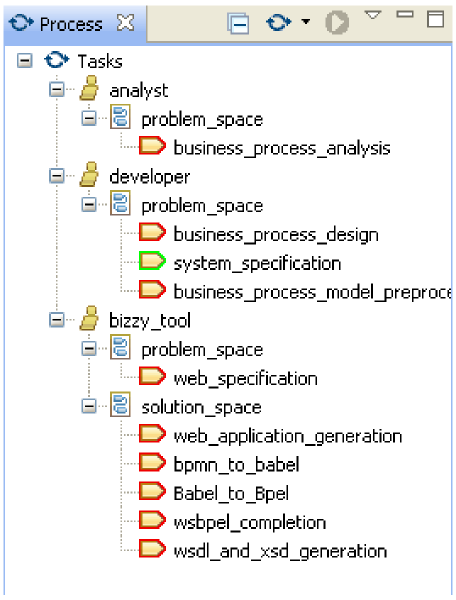
Fig. 5.7. Product configuration file

Once the export process is finished, a MOSKitt reconfiguration supporting the method is obtained. As already shown in chapter 4, these generated tools include (aside from the Eclipse plugins that support the method) a software component called the *Project Manager Component*. This component provides a series of Eclipse views that assist the software engineer during the method enactment. As an example, Fig. 5.8 and Fig. 5.9 show snapshots of the *Product Explorer* and *Process* views respectively. Specifically, in the example, the state of the running method instance is on the *system specification* task. Therefore, the *Product Explorer* view only shows the product *business process model*, and the *Process* view only shows as executable the task *system specification*.


Product Explorer

testProject
Products
problem_space
bp_model
requestBook.bpmn
requestBook.bpmn_diagram

Fig. 5.8. Product Explorer view


Process

Tasks
analyst
problem_space
business_process_analysis
developer
problem_space
business_process_design
system_specification
business_process_model_preproce
bizzy_tool
problem_space
web_specification
solution_space
web_application_generation
bpmn_to_babel
Babel_to_Bpel
wsbpel_completion
wsdl_and_xsd_generation

Fig. 5.9. Process view

5.3. Conclusions

The methodological framework and software architecture proposed in this thesis have proven successful in supporting the design and implementation of the OOWS-BP method [90]. This chapter illustrates how the three phases that compose the framework have been followed to build the method specification and to obtain a CASE tool that supports the method.

It is worth noting that the CASE tool that has been obtained provides rich support to the method, since it integrates in a seamless way all the tools that are required to support its execution and also provides guidance to the software engineer in the performance of the method tasks. For a more in-depth view on how the case study has been developed, snapshots and screencasts are available at http://users.dsic.upv.es/~vtorres/moskitt4me/.

6. Conclusions

This chapter presents the conclusions of the work introduced in this master's thesis. Specifically, section 6.1 outlines the main contributions made by the proposal. Then, section 6.2 describes how the proposal has been validated, and section 6.3 presents future work that can be carried out to extend it. Finally, the publications that have been obtained during the course of the work are listed and detailed in section 6.4.

6.1. Contributions

This section summarizes the main contributions of the present work. Specifically, this section is divided into two subsections. Section 6.1.1 describes the contributions related to the methodological framework presented in chapter 3 and section 6.1.2 the contributions related to the software architecture defined in chapter 4.

6.1.1. The methodological framework

The main contributions that the methodological framework presented in this thesis makes to the Method Engineering field are outlined below.

Model Driven Development

The methodological framework faces from a MDD perspective the design and implementation of methods. Specifically, meta-modeling and model transformations techniques are used to perform the method definition and the (semi)automatic generation of the CASE tool support.

The use of meta-modeling in the Method Engineering field is not new. Many proposals have made use of meta-models as a way of formalizing the concepts that are available during the method definition. Thereby, methods are defined as instances of a meta-model, i.e. as machine-processable models. However, these proposals do not take advantage of the possibilities offered by the MDD. The main innovation of the framework is that, contrary to current Method Engineering proposals, it leverages these models going one step further. Specifically, these models are used as input of model transformations that carry out the (semi)automatic construction of CASE tools that support the specified methods.

Method Engineering language

The methodological framework proposes the use of a standard language (i.e. the SPEM standard [87]) for the definition of software production methods.

As illustrated in chapter 2, Method Engineering proposals that make use of standard languages for method specification are still non-existent. Indeed, in [41] it is predicted that one of the likely topics for research initiatives in the next years will be a new generation of CAME tools based on internationally standardized methodology meta-models.

Method dimensions

The software production methods that are defined by means of the methodological framework cover the four dimensions of methods, i.e. the *product*, *process*, *people* and *tool* dimensions.

As illustrated in chapter 2, most of the Method Engineering proposals only cover the *product* and *process* dimensions, being the *people* and *tool* dimensions almost completely overlooked. As a result, proposals that take into account all the method dimensions together are still non-existent.

Method Engineering lifecycle

The methodological framework proposed in this thesis equally encompasses the method design and method implementation phases of the Method Engineering lifecycle.

In general, Method Engineering proposals either focus on the method design or the method implementation. The former provide adequate means for defining software production methods but very limited CASE tool generation capabilities. The latter focus on facilitating the construction of CASE tools suited to context needs, but do not provide means for defining software production methods.

6.1.2. Software architecture

Another important contribution of this work is the definition of a software architecture that proposes a series of technology-independent components that aim to support the methodological framework. Therefore, the main goal of this architecture is to propose a solution that establishes how CAME tools must be structured in order to adequately support the design and implementation of software production methods.

Furthermore, this master's thesis provides implementation details of the proposed architecture on the MOSKitt platform, whose plug-in based architecture turns it into a very suitable platform to face the Method Engineering challenges. Specifically, the MOSKitt4ME prototype has been developed. The main goal of this prototype is to eventually become a complete CAME environment that does not present the deficiencies of current CAME/metaCASE technology.

6.2. Validation of the proposal

The proposal presented in this thesis has been put into practice in order to validate it. Specifically, a case study (i.e. the OOWS-BP method [90]) has been developed in the MOSKitt4ME prototype. This prototype has successfully supported the specification of the method and the generation of the supporting CASE tool. However, this project is ambitious and, therefore, the prototype will be enhanced in the near future in order to include the obtained results in a MOSKitt released version. Thereby, the MOSKitt community (ranging from analysts to end users) will be able to test each new release of the tool and provide valuable feedback that will contribute to the improvement of the tool and the proposal presented in this thesis. Specifically, as soon as the first version of the tool is included in MOSKitt, CIT's users will make use of it in order to specify the *gvMétrica* method and build its corresponding supporting tool. This setting constitutes an adequate environment to validate the proposal since *gvMétrica* is a method that is currently being used in real projects within the context of the CIT.

6.3. Future work

The work presented in this thesis is not closed research. Different topics will be tackled in the near future in order to improve some limitations. These topics are briefly described below.

Megamodeling

During the execution of software production methods based on MDD (e.g. OOWS-BP [90]) a high number of MDD artefacts (transformations, meta-models, models, etc.) come into play. Therefore, CASE environments supporting these methods must provide mechanisms that facilitate the management of these artefacts. This hinders the already complicated task of (semi)automatically obtaining CASE tools from method specifications, one of the main goals of Method Engineering.

Megamodeling [8] is proposed as a possible solution to facilitate this task. By means of a megamodeling tool, CASE environments can manage their MDD artefacts centrally and at a high level of abstraction. Thereby, the complexity of the CASE tool generation process is significantly reduced. The main reason for this is that the implementation of this megamodeling tool is independent of the supported method and, therefore, it can be integrated into the CASE tool transparently for the generation process.

Method variability

One of the big challenges of Method Engineering takes into account the variability of methods both at modeling level and runtime [2]. Variability appears as a relevant challenge in Method Engineering, since it is very common that context changes entailing method adaptation during the progress of a project. So, mechanisms must be included in the methodological framework in order to deal with this variability. At modeling level, the use of techniques based on fragment substitution to specify this variability is proposed. These techniques permit to keep separately the common and variable parts of the method, which makes the models more legible and easier to specify and maintain. At implementation level, the introduction in MOSKitt4ME of a reconfiguration engine (e.g. MoRE [14]) is proposed. This would enable the CASE tool reconfiguration at runtime based on context changes.

Prototype enhancement

The MOSKitt4ME prototype must be enhanced so that it overcomes some limitations. For instance, the integration of a process engine such as Activiti [1] is being planned. This would provide better support to the execution of method instances.

Furthermore, a very important issue that needs to be improved is the management of the technical fragments dependencies. That is, when the method model is finished, it contains a set of technical fragments that aim at providing support to the product and tasks in the generated CASE tool. In order to adequately install the plugins contained in the fragments into the final CASE tool, their software dependencies must be resolved. For this purpose, the software dependencies must be specified within the fragments, and these dependencies must also be stored in the repository.

6.4. Publications

During the development of the work presented in this thesis, the following publications have been produced:

1. **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)

2. **Cervera, M.**, Albert, M., Torres, V., Pelechano, V., Cano, J., Bonet, B.: A Technological Framework to support Model Driven Method Engineering. 7th Taller sobre Desarrollo de Software Dirigido por Modelos (2010)

3. **Cervera, M.**, Albert, M., Torres, V., Pelechano, V.: Turning Method Engineering Support into Reality. To be published in: the 4th IFIP WG8.1 Working Conference on Method Engineering (2011)

First of all, the first publication was presented at ICSP, an International Conference ranked **A** by the Computing Research and Education (CORE) association. Specifically, this publication provides an overview of the phases that compose the methodological framework, illustrating their application by means of the OOWS-BP case study.

Then, in order to provide a more in-depth view on the method implementation phase of the framework, a second publication was elaborated and presented at DSDM, a workshop hosted at the 3rd Congreso Español de Informática (CEDI), which took place in Valencia. Specifically, this publication focuses on the application of model transformations for (semi)automating the construction of CASE tools that support software production methods, a part of the work that is of big interest for the MDD community.

Finally, the paper "Turning Method Engineering Support into Reality" has been recently accepted for presentation and publication in the proceedings of the IFIP WG 8.1 Working Conference on Method Engineering. This paper describes the methodological framework in more detail, focusing on the MDD infrastructure that supports its different phases.

References

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

[2] Armbrust, O., Katahira, M., Miyamoto, Y., Münch, J., Nakao, H., Ocampo, A.: Scoping Software Process Models - Initial Concepts and Experience from Defining Space Standards. ICSP, 160-172 (2008)

[3] Arni-Bloch, N.: Towards a CAME Tools for Situational Method Engineering. In Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications, Geneva (2001)

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

[5] Atlas Transformation Language (ATL), http://www.eclipse.org/atl/

[6] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional (2004)

[7] Bergstra, J., Jonkers, H., Obbink, J.: A Software Development Model for Method Engineering. In: Roukens J., Renuart J. (eds.) Esprit 1984: Status Report of Ongoing Work. Elsevier Science Publishers, Amsterdam, (1985)

[8] Bezivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004)

[9] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information and Software Technology, 38, 275-280, (1996)

[10] 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)

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

[12] Brinkkemper, S., Saeki, M., Harmsen, F.: A Method Engineering Language for the Description of Systems Development Methods. CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering, Springer-Verlag, 473-476 (2001)

[13] Business Process Model and Notation (BPMN). OMG Available Specification version 2.0. OMG Document Number: dtc/2010-06-05

[14] 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)

[15] Deneckère, R., Souveyet, C.: Patterns for Extending An OO Model with Temporal Features. In OOIS'98 Proceedings, C. Rolland, G. Grosz, Eds. Springer-Verlag, London, 201-218 (1998)

[16] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[17] Eclipse Modeling Project, http://www.eclipse.org/modeling/

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

[19] Englebert, V., Hainaut, J.-L.: DB-MAIN: A Next Generation Meta-CASE. Information Systems, 24, 99-112 (1999)

[20] 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)

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

[22] Fitzgerald, B., Russo, N. L., O'Kane, T.: Software development method tailoring at Motorola. Communications of the ACM, 46, 64-70 (2003)

[23] Fons, J.: OOWS: A Model Driven Method for the Development of Web Applications. Ph.D. thesis, Technical University of Valencia (2008)

[24] Glass, R. L.: Process Diversity and a Computing Old Wives'/Husbands' Tale. IEEE Software, IEEE Computer Society Press, 17, 128 (2000)

[25] Glass, R. L.: Questioning the Software Engineering Unquestionables. IEEE Software, IEEE Computer Society Press, 20, 120 (2003)

[26] Glass, R. L.: Matching methodology to problem domain. Communications of the ACM, 47, 19-21 (2004)

[27] Gonzalez-Perez, C., Henderson-Sellers, B.: A Powertype-Based Metamodelling Framework. Software and Systems Modeling, 5, 72-90 (2006)

[28] Gonzalez-Perez, C., Henderson-Sellers, B.: On the Ease of Extending a Powertype-Based Methodology Metamodel. In Meta-Modelling and Ontologies. Proceedings of the 2nd Workshop on Meta-Modelling, WOMM'06, P-96, 11-25 (2006)

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

[30] 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)

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

[32] Guzélian, G., Cauvet, C.: SO2M : Towards a Service-Oriented Approach for Method Engineering. In: the 2007 World Congress in Computer Science, Computer Engineering and Applied Computing, in Proceedings of International Conference Information and Knowledge Engineering IKE'07, Las Vegas, Nevada, USA (2007)

[33] Harmsen, F., Brinkkemper, S.: Design and Implementation of a Method Base Management System for a Situational CASE Environment. Proceedings of the Second Asia Pacific Software Engineering Conference, IEEE Computer Society, 430 (1995)

[34] Harmsen, F., Saeki, M.: Comparison of Four Method Engineering Languages. Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method engineering : Principles of Method Construction and Tool Support, Chapman & Hall, Ltd., 209-231 (1996)

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

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

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

[38] 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)

[39] Henderson-Sellers, B., Gonzalez-Perez, C., Ralyté, J.: Comparison of Method Chunks and Method Fragments for Situational Method Engineering. ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering, IEEE Computer Society, 479-488 (2008)

[40] Henderson-Sellers, B., Gonzalez-Perez, C.: Standardizing Methodology Metamodelling and Notation: An ISO Exemplar. Information Systems and e-Business Technologies, Springer Berlin Heidelberg, 5, 1-12 (2008)

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

[42] 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)

[43] Iacovelli, A., Souveyet, C., Rolland, C.: Method as a Service (MaaS). RCIS, 371-380 (2008)

[44] Isazadeh, H., Lamb, D. A.: CASE Environments and MetaCASE Tools, (1997)

[45] ISO/IEC 24744: Software Engineering: Metamodel for Development Methodologies. International Standards Organization/ International Electrotechnical Commission, Geneva (2007)

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

[47] Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc. (2003)

[48] 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)

[49] Martin, C.: MetaCASE: dream or reality. Electro'94 International Conference, 195-199 (1994)

[50] MetaCase Consulting: ABC To Metacase Technology. Technical report (1999)

[51] MetaCASE Consulting: Method Workbench User's Guide, MetaCASE Consulting, Jyväskylä, Finland (2005), http://www.metacase.com/support/40/manuals/mwb40sr2a4.pdf

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

[53] Mirbel, I.: Connecting method engineering knowledge: a community based approach. Situational Method Engineering, 176-192 (2007)

[54] MOFScript, http://www.eclipse.org/gmt/mofscript/

[55] MOSKitt, http://www.moskitt.org/

[56] Niknafs, A., Asadi, M., Abolhassani, H.: Ontology-Based Method Engineering. International Journal of Computer Science and Network Security, 7, (2007)

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

[58] 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)

[59] Odell, J.J.: Power Types. Journal of Object-Oriented Programming 7(2), 8-12 (1994)

[60] Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. Information Systems, 26(7), 507–534 (2001)

[61] Plihon, V.: MENTOR: An Environment Supporting the Construction of Methods. Asia-Pacific Software Engineering Conference, IEEE Computer Society, 0, 384 (1996)

[62] Plug-in Development Environment, http://www.eclipse.org/pde/

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

[64] Prakash, N.: On method statics and dynamics. Information Systems, Elsevier Science Ltd., 24, 613-637 (1999)

[65] Punter, H.T., Lemmen, K.: The MEMA Model: Towards a New Approach for Method Engineering. Information and Software Technology 38(4), 295-305 (1996)

[66] Ralyté, J.: Reusing Scenario Based Approaches in Requirement Engineering Methods: CREWS Method Base. DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications, IEEE Computer Society, 305 (1999)

[67] Ralyté, J., Rolland, C.: An Assembly Process Model for Method Engineering. CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering, Springer-Verlag, 267-283 (2001)

[68] 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)

[69] 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)

[70] 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)

[71] Ralyté, J., Rolland, C., Ayed, M. B.: An Approach for Evolution-Driven Method Engineering. Information Modeling Methods and Methodologies, 80-101 (2005)

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

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

[74] Rolland, C.: Modeling the Evolution of Artifacts. ICRE'94: Proceedings of the 1st International Conference on Requirements Engineering, Colorado, 216-219 (1994)

[75] Rolland, C., Souveyet, C., Moreno, M.: An approach for defining ways-of-working. Information Systems, Elsevier Science Ltd., 20, 337-359 (1995)

[76] Rolland, C., Prakash, N.: A proposal for context-specific method engineering. Proceedings of the IFIP TC8, WG8.1/8.2 working conference on Method engineering : principles of method construction and tool support, Chapman & Hall, Ltd., 191-208 (1996)

[77] Rolland, C., Plihon, V.: Using Generic Method Chunks to Generate Process Models Fragments. Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96), IEEE Computer Society, 173 (1996)

[78] Rolland, C.: A Primer For Method Engineering. Proceedings of the INFORSID Conference, 10-13 (1997)

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

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

[81] Saeki, M.: CAME : The First Step to Automated Method Engineering. In OOPSLA'03: Workshop on Process Engineering for Object-Oriented and Component-Based Development, 7-18 (2003)

[82] Schmitt, J.R.: Product Modeling in Requirements Engineering Process Modeling. IFIP TC8 International Conference on Information Systems Development Process, North Holland, (1993)

[83] Seligmann, P.S., Wijers, G. M., Sol, H.G.: Analyzing the structure of I.S. methodologies, an alternative approach. In Proc. of the 1st Dutch Conference on Information Systems, Amersfoort, The Netherlands (1989)

[84] Si-Said, S., Rolland, C., Grosz, G.: MENTOR: A Computer Aided Requirements Engineering environment. Advanced Information Systems Engineering, Springer Berlin / Heidelberg, 1080, 22-43 (1996)

[85] Slooten, K. v., Brinkkemper, S.: A Method Engineering Approach to Information Systems Development. In: Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process, North-Holland Publishing Co., 167-186 (1993)

[86] SOA Tools Platform Project (STP), http://www.eclipse.org/stp/

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

[88] ter Hofstede, A. H. M., Verhoef, T. F.: On the feasibility of situational method engineering. Information Systems, Elsevier Science Ltd., 22, 401-422 (1997)

[89] Tolvanen, J.-P.: Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. University of Jyväskylä (1998)

[90] Torres, V.: A Web Engineering Approach for the Development of Business Process-Driven Web applications. Ph.D. thesis, Technical University of Valencia (2008)

[91] Wistrand, K., Karlsson, F.: Method Components - Rationale Revealed. CAiSE, 189-201 (2004)

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