Refinement of Alternation Traces in Context of Model-Driven Change Management

Master Thesis
submitted by
Mariya Denysova

supervised by
Prof. Dr. Friedrich H. Vogt
Prof. Dr. Ralf Möller

Hamburg University of Technology
Institute of Telematics

An overview of the existing software engineering approaches for the solution of software evolution and complexity problem is executed with the focus to the Model-Driven Software Development approach. The problem of software evolution is closely related to the meta-model evolution problem. The existing related work in this area is studied and the approach of models migration when their meta-model is changed is described. The first stage of this process is the detection and classification of changes. The focus of the current work is the detection and classification of meta-model changes using tracing approach and enriching the traced information with the semantical constraints of the EMF Ecore meta-meta model. The design of a meta-model for change classification is proposed and the transformation enriching the standard EMF traces with Ecore semantics is developed.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 1.10.2007
Mariya Denysova

I would like to thank Prof. Friedrich H. Vogt, for giving me the opportunity to perform this Master Thesis under his supervision, demonstrated interest in my work and provided useful critics. I am also grateful to Boris Gruschko from SAP AG, who gave me such an interesting topic for my research. Separately I want to thank Simon Zambrovski, who was not only a technical adviser for me during this half a year, but a close friend, teacher, co-thinker and strict critic. His explanations, patience, readiness to hear and evaluate even my wrong ideas, turned these six month not only to efficient work, but also to a real pleasure.

Contents

List of Figures

Chapter 1

Introduction

This is a Thesis on Software Engineering. This means that however specific the task described here is, the main value of the current work is another step towards the solution of the general software engineering problem: to shift the software development process from the world of unpredictability, creativeness and constantly increasing system disorder to the world of strict engineering rules, laws, strategies and structures.

The term "Software Engineering" was first popularized in 1968 during the NATO Software Engineering Conference (held in Garmisch, Germany) and has been in widespread use since. The official report from this conference introduced the main notions and challenges of Software Engineering like Software Requirements, Design Process, Problems of Scale, Reliability and Performance, Software Maintenance and Evaluation. Since then the computer scientists all over the world were arguing, if the process of software development is a real engineering process. One of the well know opponents of Software Engineering was Donald Knuth, who insisted that programming is an art. Still after 40 years of intensive research in this area and increasing growing amount of the commercial software development it is an opened question.

The need for building an engineering science around the software development process arose from the problem of constantly increasing size and complexity of the software systems. To illustrate the problem let's consider the process of software development from the point of view of an art. A skilled and talented software developer can come with a genius piece of software as a talented painter can paint an outstanding paint. Now lets imagine that we need to develop a great complex software system. To do that we would hire 200 skilled programmers. What would be their result? Approximately the same as in case of hiring 200 hundred talented painters to paint a picture with the size of the area of Vatican. Unless 200 painters would agree on the strict working strategies, style of painting, used colors etc. the result would be unpredictable and bad. People usually speak of genius artists as bright individuals, whereas the successful team work is possible only under the strictly established process.

After 40 years the role of Software Engineering as an engineering process is commonly understood, Software Engineering methodology evolved and matured, yet not everywhere accepted as a real science (e.g. Russia). Software processes and approaches give tremendous support for the commercial software development. Still Software Engineering remains quite fuzzy and non-precise, having lots of the questions unanswered. Again and again new software development projects fail to meet time and quality requirements, software systems evolve with more and more increasing complexity. Software development projects experience all problems and challenges of innovative projects like unpredictability, creativeness and high risks, but also has

to solve its own, unique for software area difficulties. One of them is the Software Evolution problem.

Nowadays the software systems support every business area. The laws of market economy prescribe constantly changing and self regulating nature of the businesses on the market. The market situation influences each separate company, there for each company have to constantly adjust its business activity to the market needs. Therefore the software systems, supporting each business have to be constantly changed.

But why these changes bring constantly increasing complexity to the software systems supporting the businesses, if they don't bring it the the businesses themselves.

1.1 Motivation: Software Evolution Causes Software Complexity

From the first site the most cost consuming part of the software development is the stage of the software creation. Once the software has been created it could be easily deployed for the customers almost for free. Unfortunately nowadays it is commonly understood, that the most expensive is the maintenance of the software in the sense of the constant adjustments to the new business environment. According to [Pre00], industry data indicate that between 60 and 80 percent of all effort expended on software is expended after this software is delivered to the customer for the first time and the substantial part of these costs is caused by the side effects of the introduced software changes. These side effects occur because due to the interconnections in the system each change almost always cannot be treated separately and has its influence to the whole system resulting to the need of new system adjustments. This shows that big amount of interconnections, i.e. high level of coupling between structural elements of a system, results to the lower maintainability of a system. As it is shown in [McM95, Mey88] modularity and loose coupling are important principles that allow to improve of software maintainability and manageability. Module is treated here as the basic unit of decomposition of software systems, irrespectively of its actual type: class, component, or their combination.

Therefore, the level of "coupling" in the system expresses one of the aspect of system complexity, namely its *structural (architectural) complexity*, which is one of the approaches for assessing the overall complexity of a system architecture [Pre00, LB06]. Generally, the term "complexity" may include much more system characteristics like computational complexity (algorithmic), logical complexity etc. (see [LB06, Pre00, Goo04, McM95, Fen94]). In the current work, when speaking of complexity, we imply mainly the structural complexity.

The reason, of the constantly growing complexity of a software system lies in the digital nature of software. Namely, we can distinguish the following problems:

- The first reason originates from the fact that there is no known way to create error-free software. Every time the software is changed these mistakes are accumulating.

- Another reason has pure sociological nature. Because the cost of an already written code almost equals to zero, people are inclined to copy or live the old code and add additional features to it, instead of throwing it away and replacing with the new one. There, the old mistakes are not replaced by the possible new ones, but the new ones are added to the old ones.

  It is quite obvious, why this issue only applies for Software Engineering. If we were producing automobiles, then in case of invention of a new better technology

for producing car engines, we would not start the production of cars, which would have two different engines inside. We would delete everything from our new car model that is not needed, because the to produce not needed parts requires material and working hours. In case of software development to leave unneeded parts is easier then to delete them, especially if it is not clear, which parts are really not needed.

- One more reason reason is that each next version of a system have to hold and operate with all the previously collected data as well as working with the new data, probably with the changed data structure.

  So in case a car producing business is switched on to the production of new car engines, the support for production of old car engines usually remains in the software system.

The first research attempts on the topic of Software Evolution were undertaken almost simultaneously with the birth of Software Engineering. The pioneer in this area is Meir Manny Lehman, who documented the evolutionary characteristics of the mainframe operating system, OS/360 to IBM in 1969 [Leh69] and later, in 1985, expanded the study to other programs in his book "Program Evolution: Processes of Software Change"[LB85]. In these early works Lehman formulates his first three laws of software evolution, which were later extended to 8 [Leh96]. The general idea of Lehman's laws is that a software system has to be progressively changed to satisfy users needs which in its turn cause the constant increasing of system disorder.

Back then in the 80s the main effort was put to answer the question: "How software evolves, what are the dynamics and the laws of increasing software complexity?" Nowadays the focus has changed. The Software Evolution studies became a Software Engineering issues and are mostly focusing on the software change management questions, i.e.: "Which approaches can efficiently slow down the increasing complexity of evolving systems?".

1.2 Approaches to Solve the Software Evolution and Complexity Problem

Lehman was looking for the answer to this question in the software development process. Current software development processes like Rational Unified Process [Kru03, JBR99], Waterfall and Iterative Programming as well as modern Agile Methods among other issues target to solve also the Software Evolution problem. All these approaches pay attention to the importance of modeling techniques as an analogy to the modeling in the general engineering science. These modeling techniques brought to life sketches, diagrams, drawings and specifications of different kinds, which were meant to become an instrument to share knowledge among the team members and to transfer it between software phases. This also required the corresponding modeling rules, modeling languages as well as their tool support.

Another approach to solve the Software Evolution problem was to decompose and structure the software code itself [Jac86]. Modules, classes, components and frameworks all these well known structural software artifacts arose from this approach. The next step in this direction brought to the idea that to decrease the entrophy and make the code less error-prone it is reasonable not to hand write and copy/paste but to generate the routine, repetitive code, thus making the separation between the application logic code and the infrastructure code.

The combination of modeling and code generation techniques resulted into Model Driven Software Development approach, where the model and the code are the two

perspectives to the same software program. Therefore, the problem of software evolution implies the model evolution issues.

The current Thesis will give the explanation of the advantages of the Model Driven Software Development comparing to other software development approaches with respect to the Software Evolution Problem. It will also discuss the different aspects of Model Driven Software Evolution and will give a general approach towards the solution of the Model Evolution problem.

1.3 Scope of this Thesis

This thesis is a thesis about the application of Software Engineering methodologies for the development of large enterprise software systems. The methodologies described in this thesis, and the conclusions made from the discussed approaches are done under this assumption, and may not hold in other domain areas.

Chapter 2

Motivation for Model-Driven Software Development (MDSD): Predecessors, Problems, Relation to Software Complexity

As it was already said, Software Engineering is a science about the methodologies applied to increase the speed and quality of software development as well as to decrease the development cost. One constraint on the way to achieving these goals is the Software Evolution problem. In this chapter we will discuss the existing trends in the Software Engineering and their relation to the problem of Software Evolution.

2.1 Software Development Process Perspective

All the development methodologies aim at structuring the working process in the development team, establishing the workflow scenarios, improving the communication and information exchange, therefore, gaining the positive synergy from the teamwork.

Naturally, the software development process resemble a general problem solving loop [Rac95]:

1. Fist, the problem is defined, which is usually done on the stage of requirements definition and analysis.
2. Then the solution is developed, which corresponds to the design phase.
3. The solution is produced and integrated (development and integration phases).
4. The result is archived and analyzed.

After this four stages the problem seems to be solved. However, usually after the result is achieved, its analysis brings to the detection of related problems, the problem is extended, refined or generalized and the loop is repeated.

Therefore, all the development methodologies have these stages in common and differentiate mostly in the ways for their composition, iteration, exact deliverables produced on each of them and the approaches of knowledge transfer in between.

We can distinguish between the traditional development processes and modern Agile approaches.

2.1.1 Traditional development process

Historically, the first model of the software development process was the *waterfall model*, a model of the sequential software development [Roy70]. Abstraction is the key mechanism in this model. The early stages of the development process are the stages where the system is considered on the higher abstraction level. From analysis through design and to the development phase the system is being constantly refined. The abstract representation of a system is expressed in form of natural language (specifications and other documentation) or a modeling language (graphical representation). Abstract modeling of a system managed to bring better clarity to the system structure.

Unfortunately, this development process was completely uncapable of adjusting to the changing requirements to the software. The natural solution in this case was to repeat the whole process, which brought to the invention of the iterative models.

The iterative development process is considered to be more efficient, because in most cases it is impossible to conduct the initial fundamental analysis of the whole problem space, to define and foresee all the related problems and their effects and to build the corresponding design decisions.

Another disadvantage of the waterfall process is that users see and are able to evaluate the developed software only after the development is finished. The user assessment and suggestions cannot be taken into account during the development process. Rapid prototype creation helps getting the relevant user feedback.

Two extreme cases in the development process can take place:

- The waterfall development, when the attempt is made to design the whole system on the first iteration, which is not successful in most cases and the "fear of unanticipated requirements often leads to overengineering" [Eva03].

- The other extreme, when the whole development is an endless series of prototypes. Each such prototype is "the simplest thing that could possibly work" [Bec00], which is iteratively refactored and refined. This can bring to the situation when "the attempt to avoid overengineering can develop into a fear of doing any deep design thinking at all"[Eva03] and often results to a messy unstructured solution.

Therefore, the type of a problem, the level of uncertainty and risks should be considered to balance successfully the actual development process between these two extremes. An appropriate adjustment of iterative methodology and tool support can also shift this balance towards rapid prototyping development.

The form of iterative development, when the waterfall model is simply repeated, proved to be uncapable of reacting to fast changes to the system requirements. Changes were often much more easier to be introduced on the source code level, without any need to to clarify the picture on the more abstract model level. Therefore, only the development stage was often involved into iterative process. The need to update models or write documentation became just an overhead and was often omitted. This resulted to the gap between the models and the source code.

The need for the efficient methodological and tool support of iterative development resulted to the invention of the new, agile software development methodologies.

2.1.2 Agile software development methodologies

If the traditional development processes aim to the decomposition of the development process, assigning responsibilities to each team member and the information flow ("communication") between the developers by means of documentation, the main focus of agile methods is the best integration of the developers in the whole development "game". This is done by refusing from writing much documentation for code clarification and organizing personal communication inside the team in forms of regular meetings, discussions and seminars, using a blackboard and a piece of chalk. This promised to guarantee, that only cornerstone questions are discussed and modeled instead of writing documentation which may appear to be never used by anybody.

An important role in Agile methods is played by the the software code itself. After the overhead nature of models and documentation in some projects was clearly understood, the importance of program code itself increased. The developers realized the need and usefulness of a step from high-level abstract models down to source code paradigms. One of the reasons, why this step became possible, was the recognized usefulness of programming paradigms called "patterns" [GHJV95], which are general purpose source code abstractions. Their semantics is general, commonly known by all developers and independent of the application domain and developed system.

2.2 Modeling Approaches

As it was said before an important software design mechanism is abstraction. The result of applying abstraction mechanism is expressed in models, which help to neglect the irrelevant details to understand the system structure. Models are the artificial temporary system simplifications, which are further refined. Often this refinement is merely a routine, which is desired to be automated.

The process of modeling is always a process of finding proper abstractions. The question here always is: what is a "propper" abstraction? Unfortunately, there could be no formal way to define the term "proper abstraction" as well as there is no strict rule on how to find them.

The capability of making abstractions is a unique property of human mind and is expressed through our natural human language. No animal as well as no computer is able to do that. How the human mind makes "proper" abstractions is the focus of studies of Artificial Intelligence field. Until the scientists of this field do not understand this mechanism, no formal laws on how to do good abstractions could be done. Still different practitioners managed to work out some recommendations on this topic, often referring to modeling as art [Lie06, Eva03].

The software engineers classify modeling by the object of modeling as following:

software modeling - the abstract representation of the software system used by engineers during the development process;

problem modeling - is the collection of software requirements as described by the customer of the software system or its future users, expressed in form of a specific professional language, using domain specific notions.

2.2.1 Software Modeling

The software modeling is the abstract representation of the solution to be developed. In this sense it is similar to the design plans or construction plans used in all other engineering disciplines.

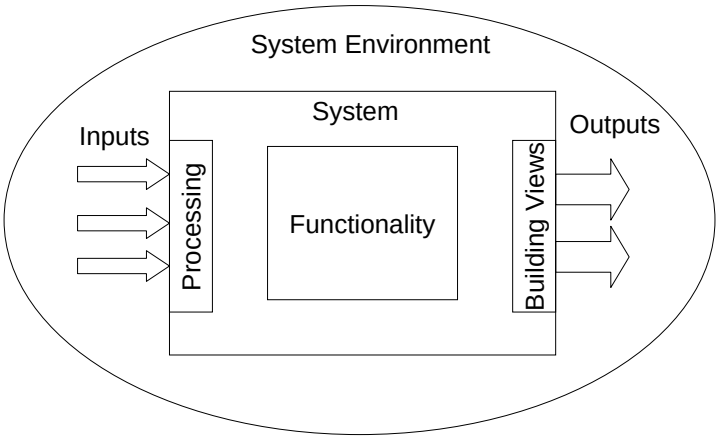

Figure 2.1: General Structure of a Software System

The software modeling is closely related to the aspects and building blocks of software architectures. The different types of the software architectures are relatively well understood. Therefore, the methodology for its modeling can be done.

Historically the problem of software modeling on the first stage developed towards the unification of modeling languages and modeling process. The peak of this process was the invention and popularization of Unified Modeling Language (UML), which "is as general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system" [RJB99]. This means, that UML is a language for modeling all kinds of software artifacts, and possess required expressiveness for that.

The standardization of UML facilitated the communication among the developers, because this language was well understood by everybody. Every developer could read it and make the design using it. Because UML was a common language for expressing software abstractions, different tools (UML-tools) were developed, capable to generate source code skeletons out of UML documents often called Computer-aided software engineering tools (CASE-tools).

2.2.2 Problem Modeling

The functional software requirements can be considered as problem descriptions, that must be solved by the software.

On one hand, to describe these requirements the specific professional language (*domain language*) is used by the business people working in these business environment (*domain experts*). The term "domain" is often assigned somewhat different meanings by different researchers and practitioners. The commonality of all definitions can be expressed as following [BRJ05]:

Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

On the other hand any software system is an open system as defined by the General System Theory [Bar93] and is a part of a larger business system (see figure 2.1). This means that a software system should fit into the business environment, i.e have knowledge about the structure of this business system, as well as to fulfill

definite functional requirements, that depend on information inputs from business environment and influence this business environment with some output actions.

Therefore, software requirements are defined with respect to the actual business processes that exist in the given business system. Consequently, the models of the existing *business environment/business processes* are needed as instruments to express these requirements. These models are usually called *domain models*, and the process of building such models is called *domain modeling*.

So, domain models are the instruments to express the sets of software requirements as well as the basis for building the software solution, which has to implement these requirements and have knowledge about the business environment of the software systems.

Unlike the software modeling that express abstractions of software artifacts the problem modeling is the modeling of the business environment and the software requirements that are inputs and outputs to the software system. The problem modeling is usually done by the business experts who express the problems, they want to solve by the software system, while the software modeling is done by the software developers, who want to express, how they are going to solve these problems. To some extent, software modeling is a domain modeling, where the domain are the different architectural software artifacts and the software developers are the domain experts. Therefore, we can say that software modeling covers only one domain area and, consequently, is easier and better understood, than modeling of the verity of other domains.

The area of different problems domains is wide. Each software solution innovative, because it is done for a special, unique, problem. Therefore good recommendations for the doing successful domain-specific abstractions are much more difficult to be done and modeling of a domain area is resembles more the notion of art. Domain-driven design is an approach to develop software systems when the domain model plays the central role in the whole development process.

Domain Specific Languages (DSLs)

In the section 1.1 the term software complexity was introduced, as well as some reasons causing the software complexity. One of the reasons to for the software to be complex, is the complexity of the domain for which this software is built.

Among lots of the definitions of complexity some define complexity, to be equivalent to the amount of the information required for the description of the complex object or system. The linguists give the following definition of complexity [Här69]:

Complexity: the property of a system or a model, that makes descriptions of the complete behavior in a random (or in any) language difficult, even if the information of all components and it's relations is available.

According to this definition not every language is capable of describing complex domain in the best possible way. UML was quite useful for abstract modeling of software artifacts, but appeared to be quite inconvenient for describing the complex domains precisely, clearly and briefly. The Object Management Group (OMG), the UML standardization organization suggests to use UML profiles to express as the embedded domain-specific language to express the domain-specific concepts. Another possible way is to use separate domain-specific languages (DSL's) for domain modeling. A well defined DSL should require less information for describing the domain models and have to be intuitively understood by the domain experts.

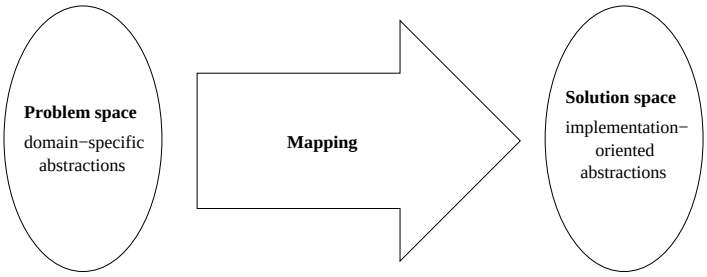

Figure 2.2: Mapping between problem space and solution space [CE00]

This conclusion defined a new modeling trend: a shift from general purpose modeling languages into the area of domain specific modeling, using domain specific modeling languages.

2.2.3 Mapping between Domain and Solution Spaces

From the point of view of the software system developers the actual business environment together with the requirements to the software operating in this environment can be referred as the *problem space*. Then, the development of a software system can be considered as finding a proper mapping between the problem space (i.e. the application domain and the software requirements) and the solution space (the developed software) as shown in figure 2.2.

In the traditional software development process the mapping from the problem domain to the software domain is usually done manually and is the central part of the development effort. Often the domain model is not build at all, and exist only in form of the requirements specifications, or even worse, in some form of half-couscous desires of the customers. The process of mapping in this case is the process of building the system design form the requirements specification (or oral customer's explanations) using some general purpose modeling language (in case of object oriented approach usually UML).

This process is generally intuitive, error prone and individual for every developer. General recommendations exist on how to solve this task, e.g. the design patterns as a set of standard solutions for standard problems. In case of identical or similar domains a good variant of this mapping is often also identical or the similar.

Similarly to the CASE tools for generating program code from UML models, the corresponding tools for DSL's were desired. The powerfulness of such tools (as well as their complexity) would be higher comparing to the UML-tools because such tools had to be able to perform the mapping from any domain to the corresponding solution space. Using such tools by all the developers in a special domain, would provide unified and optimized variants for such a problem/solution mappings. Such tool could be successfully compared to a domain-oriented design-pattens cookbook developed for a definite domain and stored not in a paper form but in a digital form and containing not only the design decisions, but the source code of such decisions too.

However despite the obvious usefulness of such tools their development requires much effort. Because these tools are designed to perform mapping from any problem to the corresponding solution space, their designers should develop the whole range of prefabricated solutions (general solution space) and prescribe the rules of how to decrease these solution space to only one solution, depending on the description of

the given problem. The design of the mapping rules between problem and solution spaces can be done in a form of a general purpose modeling language (e.g. UML), or in a form of a DSL, design specially for describing mappings.

Another problem exist, because every DSL describes only one special domain out of many, therefore for every DSL a separate tool had to be developed.

Therefore, the software engineers proposed to build some generic tools, capable of generating source code, based on a model expressed in a definite DSL and another model, which models this definite DSL, which could be expressed using some standard general purpose modeling language like UML.

2.3 Software Decomposition Perspective

As was discussed before, system evolution can increase system complexity. To decrease complexity special counteractions have to be done (see section 1.1 and Lehman's second law [LB85]).

What is the general meaning of system complexity? It is a system featuring a large number of interacting components. Therefore, the natural way to decrease system structural complexity is:

- the decomposition to subsystems, so that the internal coupling inside subsystems is higher, than external coupling between the subsystems;
- and the encapsulation of subsystem implementation and hiding it behind the interfaces, which are the only instruments through which other subsystems can use the functionality of the given subsystem.

Such subsystems are sometimes referred "structure points" [Pre00, McM95] or simply as "modules" or "components" [Mey88]. Large number of software engineering approaches are devoted to the studies of approaches for software decomposition methods which can decrease perceived complexity of the system and increase its understandability.

While modeling is a temporary decrease of the system complexity for better human understandability, decomposition really decreases the complexity of the end system.

2.3.1 Component Technologies

The history of software development knows a lot of attempts of system and source code decomposition into functions, modules, files and later to classes and interfaces until the notion of "component" was finally introduced. Comparing to modules components are meant to be exchangeable. This means, that one component can be used in different applications and one application can easily exchange its one component to another.

Therefore changes to the system can be introduced by complete exchange of a component, instead of introducing new changes to an existing code. This approach of introducing changes does not increase the system complexity.

The possibility to assemble an application from general purpose interchangeable components is the main challenge of the generic programming, which in extreme case could completely replace programming. The configuration of components could be expressed in some configuration language. The practical use for this ideas can be seen in construction of template libraries (e.g. in C++) or generics in Java, where instead of using a new configuration language an extension to existing language was done.

2.3.2 Frameworks

Framework is a prefabricated combination of software components, that can be reused across many similar applications. This combination of components is usually already structured in a form of a certain design decision. Therefore, framework is usually defined as a reusable design decision. In this way frameworks resemble the design patterns with the difference that the latter imply the reuse of the design solution only and the former incorporate also the code reuse. Another difference is that frameworks correspond to a larger grained design decisions on a higher abstraction level, i.e. they are usually meant for a definite problem domain, or a family of applications and the design patterns for a definite subproblem area inside such a domain. Therefore frameworks often consist of combinations of deign patterns.

So generally speaking, a framework is all the program code prefabricated and reused across similar applications. Lets analyze which code is usually reused in a software system.

As shown earlier in figure 2.1 the designed system functionality depends on the inputs from the system environment. The result of the work of the system influences the system environment by its outputs. Therefore, every software system consist of following subsystems:

- A subsystem or mechanism for processing input control signals or data to the system from the system environment.
- A subsystem or mechanism for preparing output control signals or data to the system environment.
- A subsystem or mechanism for modeling the system environment, which changes its state and notifies the system by the input signals.
- A subsystem implementing system functionality or application logic.

Usually the subsystems for processing input and output signals are independent of the exact application domain, but are strongly dependent on the used technology. A lot of frameworks provide a substantial support of these tasks. The frameworks which depend only on the technological solution, and make no assumptions about the application domain structure, i.e. system environment, can be called domain independent frameworks. On the other hand in every framework we can extract a domain-independent part (see figure 2.3). This can be considered as the application of the separation of concerns principle.

The system environment is very strongly dependent and usually needs a separate manual design and implementation for each application or domain. Common software and domain modeling techniques are usually used for that (see section 2.2).

The usage convenience of the frameworks is achieved due to the combination of the following components:

- Abstract classes, proving for the extensibility of the frameworks by subclassing, utilizing the fact that it is easier to subclass an abstract class then to create a new one. This is a property of so called “white-box frameworks”, which are easily extensible, but require a good knowledge of the framework design.
- Interfaces, behind which the implementation of routine functionality is hidden. This mechanism is common for so called “black-box frameworks” and is usually preferred for commercial framework products.
- Application logic control, programmable and partly configurable.

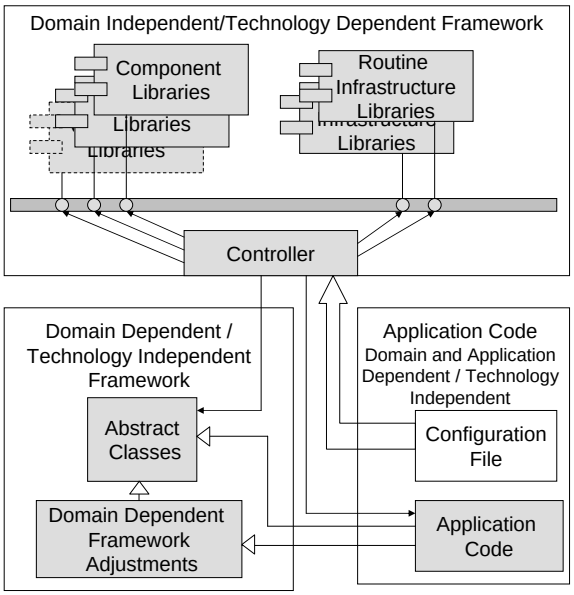

Figure 2.3: General Structure of a Framework

- Component library holding the components (also classes or functions) that could be optionally used in an application.
- Extension points for plugins.

After the usefulness of frameworks was understood the development activity split into two forms: framework development and application development. Often this split occurred not only among the software development companies, but also inside a company among development teams.

A framework provides a prefabricated solution design, a template, i.e. it exist in solution space. The domain dependant components of frameworks provide prefabricated implementations of domain entities, therefore cover some part of the domain space and it's mapping to the solution space. Still this support is almost never complete. Therefore the problem of increasing software complexity during the system evolution still exist, when using frameworks. The richer the domain specific part of a framework is, the easier this problem can be solved.

2.3.3 Generative Programming

Some stand alone components (i.e those that are sold and shipped as independent software products) or the components that are parts of frameworks that are meant to be used in different domains have much in common. The difference can be extracted in form of some configuration data. These are the configurable components. Such components accept the configuration parameters and according to this parameters dispatch the calls to the other framework components. In this way the application logic can be partially programed.

The process of dispatching the program calls is not a very strait forward. A lot of checks have to be done in the runtime. A more efficient way is to generate a version of a component based on configuration parameters. In this way the expensive checks are

done only once, by the generation of a component. This corresponds to the tunning of a component to a special domain or a special application inside this domain. The further operation of a component is based on this specially tuned (generated) version of a component, avoiding unnecessary checks and computation complexity.

The components responsible for the generation of other components and capable of processing some input data (parameters) to perform generation were named generators. This input data can be complex and structured, depending on the level of flexibility of the output, the generator is supposed to produce. Often the structure of the input data to the generator of domain-dependent components resembles the domain structure, i.e. the generators are build to process the domain models, described using definite DSL's.

By changing the input data to the generator the changes to the software system can be introduced. This changes are not influencing the generator structure, therefore they don't influence the complexity of mapping from domain to solution space. Therefore, the complexity of a system does not increase after the regeneration. In fact this approach equals to the "generation" of a new software system, but with the changed requirements.

2.4 Model-Driven Software Development

Model-Driven Software Development approach evolved as a combination of modeling techniques and the generative programming. It evolved from a slightly older approach: generative programming.

In the late 90 software engineers all over the world began to loose their way in different existing methodologies. Traditionally they were supporting the Object-Oriented approach, but steadily gathered different beyond-the-objects techniques: Domain Modeling, Component Orientation, Aspect-Oriented and Intentional Programming. Until, finally, in 1999 on the Firs International Symposium on Generative and Component-Based Software Engineering (GCSE '99) in Erfurt, Germany, the generative programming was born, as a methodology for combination of all these methods [CE00].

The generative programming offers a structure for a successful interplay of domain models written using DSLs, which are inputs for the generators, capable of refining and transforming them to actual generated software components, which in their turn interact with some other generic components in a form of a framework, which due to the generated components is specific for all applications in one application domain.

Although this process was initially defined withing the generative programming approach, the term generative programming is usually understood in a much broader sense. Often generative programming is considered to be any kind of software development, where some parts of code are generated depending on some input. Generative programming also does not obligatory prescribe the use of the whole combination of DSLs, generic components and generators.

To avoid these misleading terminology very soon a Model-Driven Development (MDD) or more precisely Model-Driven Software Development (MDSD) approach evolved, which prescribe the domain models described in DSLs and code generation as obligatory parts of the development process.

A well known and commonly used term Model Driven Architecture (MDA) is a Object Management Group (OMG) effort towards achieving the standardization in MDSD approach for interoperability of platforms, tools as well as the developed applications. The restrictions set by OMG to achieve this goals often decrease the flexibility of individual applications and increase the development effort. Therefore

MDA recommendations are commercially not very common and are not influencing the ideas of the current Thesis.

2.4.1 Frameworks and Applications

An MDSD Framework is a special kind of framework that obligatory includes some generic components (platform), a DSL for model definition, some kind of parser that can read models, and generators that can generate other models or application code.

The work of application developers is the definition of a model, running the generator and addition some specific application logic to the generated code. In more complex tasks the models have to be transformed, enriched with some details or merged with other models before the actual program code can be generated. These transformations are defined using special transformation languages and are then executed by the generators.

The simplification of the job of application developers shifts more tasks to the framework developers, who are responsible for the definition of DSLs, developing convenient editors for model creations, writing libraries containing important transformations. Although this work is usually done by another team of the developers, then the application development, still these tasks are usually quite domain specific and are done within the same company as the application development.

There usually is a third group of developers who solve even more general domain independent tasks as developing the generators, editors and tools for DSLs definitions. These tasks are currently solved by proprietary and open source framework developers.

The scope of a current thesis refers to the third group of tasks as it is the development of a tool capable to migrate models in case the DSL is changed.

2.4.2 Model-Driven Change Management

MDSD approach promised to have a good support for the iterative development and change management. From the first glance the change management really looks simple.

The changes to the MDSD application can occur in two places:

- changes to the model, i.e. structural changes of the modeled objects. In this cases the abstract model is changed and the application is regenerated; no complexity increase occurs;
- changes to the application logic, i.e. those changes that are very application-specific and are introduced not the generated infrastructure code, but to the manually written code. Obviously in the MDSD application such code exist, but usually it is only around 40% of code [SVC06], and such code is usually "component specific", hidden behind a good component interface, which is generated and model dependent. So even mistakes or bad manual component implementation or change will influence only one component and will not increase the system complexity.

However the problems would occur in case of changes not in the application itself, but one level up, in case of changes in the framework. One such change can influence a lot of applications that use this framework.

This is a general software engineering problem, or can be even extended to a general engineering problem. What happens if the tools used for any kind of engineering development are changed. Generally, a change to a more general engineering tool or methodology causes the need for migration of all the depending specific activities.

More specifically in the software development migration is usually needed when the implementation of language compilers changes, the component libraries or frameworks functionalities are modified.

Often as a solution for such problems the backward compatibility principle is used. That is, the changes to the libraries are just added, and the earlier versions remain as part of the library. The recommendations are done on the usability of introduced methods, functions or components, still the corresponding earlier variants are not completely forbidden, but just not recommended, often marked as "deprecated".

Another work around could be the usage of very generic interfaces for calling the library functions. The substantial changes to their implementation are not invalidating the client code of such libraries. The same is a common practice for building frameworks, e.g. Eclipse.

Anyway these approaches are increasing the implementation complexity of such libraries and frameworks, and are useful mostly for commercial widely used products with large number of clients.

The domain-specific part of MDSD framework is used only within one domain. Therefore have less users then domain independent part. The domain dependent part is very much change sensitive, because should be able to fast react to the system environment changes, i.e. domain changes. Therefore, the two approaches described earlier are not very useful.

The main part of the domain specific part of MDSD framework is namely the DSL (and the domain-specific solution code generated correspondingly to this DSL). If a change of a DSL is done, the models which were written using the old version of a DSL may become invalid with respect to the new version of the DSL. These type of changes within MDSD architecture are the topic of this Master Thesis.

2.5 Motivation: Why SAP Goes for MDSD?

If whole chapter up to now can be treated as the motivation for MDSD approach, this section explains why MDSD approach is important for the SAP AG, where my Master Thesis was done.

SAP is the third world largest company in the world and the largest developer of ERP systems worldwide. This means, that the purpose of the software developed at SAP is to provide solutions for the complex support of various businesses. Naturally any SAP software solution installed in an organization complies to the software system notion introduced in section 2.3.2.

The SAP ERP system resembles the notion of framework, which can be composed of different components, have a flexible mechanism to perform tunning of the system for the specific needs of an organization and provide adjustments by adding client specific code, usually written in ABAP proprietary language used for SAP systems.

Such customer specific solutions are build using SAP platform (usually SAP R3, in the future NetWeaver), and add-on applications and application components. To build this structure a deep analysis of the organization business environment and the customer business processes have to be done.

Depending on the customers business area and the application or component which needs to be adjusted or reprogramed (or even regenerated) the modeling was done using a special domain oriented modeling language. Different teams working on different SAP components and with different customers being aware of usefulness of domain modeling invented them as well as their tool support. In most cases, being very powerful with respect to the intended use case, these tools showed deficiencies concerning interoperability [AHK06].

A common infrastructure, i.e. a framework, providing the common tools for defining DSL, storing and displaying the models developed using these DSLs was needed. First this initiative started as a research project at SAP Research laboratories, and later after the commercial usefulness of such tools was understood, in the middle of 2005, SAP launched "Modeling Infrastructure" (MOIN), a development project as part of NetWeaver application platform. The official goal of the MOIN project is to implement the platform for SAPs next generation of modeling tools.

Although the current work was done in the MOIN team, the platform for the current proposal on how to manage DSL changes was done using Eclipse Modeling Platform (EMF) [EMF], an open source MDSD platform from Eclipse community. The reason, why EMF was preferred to MOIN, is that EMF is currently a released framework, while MOIN is still in the development stage. Some other reasons will be explained later. Still, due to the similarity of the tasks that are meant to be solved by EMF and MOIN, the results of the practical work conducted using EMF can be important for the future MOIN design decisions, as well as in any other MDSD framework.

2.6 Summary

Software engineering as any applied science is a science about a trade off. The main idea that goes through the whole chapter is that Software Engineering evolved as a science about the methodologies for the simplification of the software development. The traditional development processes appeared to be incapable to adjust to numerous changes to the software systems requirements and to be efficient in case of fussy and unclear initial conditions. Therefore the tools and methodologies evolved which efficiently support the iterative development not breaking the software architecture and without bringing additional complexity to the developed software.

A promising technology for major software enterprise applications projects is currently MDSD approach, which has convenient methods to support iterative development due to the possibility for abstract domain modeling and refinement of the model to the source code automatically using an application generator.

This approach talks about the methodology for the effective development of the families of software corresponding to one domain. Unfortunately, it has a challenging problem not solved up to now: how to organize automatic migration of each application, in case the generic components, supporting the common infrastructure functionality, is changed.

The more specific problem is how to migrate application models in case the DSL for the description of these models is changed. This problem is not solved up to now, more about it in chapter 4.

Chapter 3

Model-Driven Software Development: Notions and Concepts

3.1 Definition of Main Notions

After the introduction to the Model Driven Software Development is done and its purposes are clearly understood the required notions and definitions used in MDSD will be introduced.

The notions, their definitions and the terminology used in the current work are is based on the book of Thomas Stahl and Markus Völter "Model-Driven Software Development" [SVC06], if not otherwise specified.

3.1.1 Model-Driven Design

Firstly, the notions for domain analysis and description (used during the analysis and design stages of the software development process) will be introduced. These concepts and their relations are depicted in figure 3.1.

Domain : The starting point of MDSD is a *domain*, which is a bounded field of interest or knowledge usually with its own special concepts and their terminology (defined in section 2.2.2). The domain is extracted out of the real world as an abstraction of some its part. The sign of a good abstraction is the easiness to assign a name to the domain. After the separation of domain it can be analyzed, the core concepts and their relationships can be extracted. This process resembles the projection of a "piece of world" to the formalized IT plane.

Model : Model is the formalized representation of a "piece of the real world" and a definite state of the things in it. It represents one of the possible domain states. A model is a "sentence" formulated in modeling language and obtains its meaning from the language semantics.

Meta-Model (MM) : In the context of MDSD, it is absolutely mandatory to clarify the structure of domain (i.e. its ontology), so that it is possible to formalize this structure or its relevant parts. This formalization takes place in the form of *meta-model*. The meta-model contain the description of the *abstract syntax*

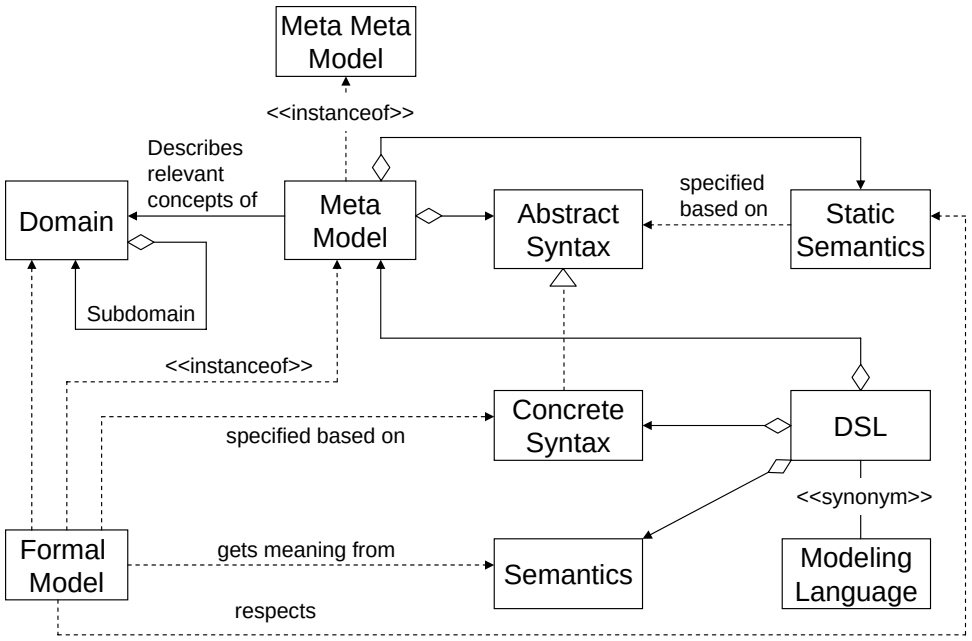

Figure 3.1: Concept formation: modeling and DSLs [SVC06]

and the *static semantics* of a modeling language (i.e. DSL). A meta-model describes concepts that can be used for describing the state of a domain by creating models.

Meta-Meta-Model : A meta-model must itself have a meta-model that defines the concepts available for meta-modeling. This is the role of the *meta-meta-model*. meta-meta-models are important in two respects: for people defining meta-models, it defines the language that is used for doing so. For framework developers, the meta-meta-model is the basis for integration among meta-models, i.e. the basis for integrating domain specific framework into the domain independent frameworks.

Abstract Syntax : represents the abstract concepts of the formalized domain and their relationships, irrespectively of their actual form of visual representation. Several concrete syntaxes can correspond to one abstract syntax.

Concrete Syntax : specifies the concrete notation (textual or graphical). Several concrete syntaxes can correspond to one abstract syntax.

Static Semantics : The static semantics of a language determine its criteria for well-formedness, i.e. precsribes the rules for the combination of meta-model elements. Violation of these rules invalidates a model. In the current work by using the word "semantics" the static semantics is meant.

Dynamic Semantics : Although the semantics of a DSL cannot be formalized it has to be commonly and clearly understood to all people working with this DSL. If the DSL's semantics strictly corresponds to the actual domain, it can be intuitively clear for domain specialists. On the other hand it should be uniformly understood by the developers that develop the software solutions in the given domain. Often the extensive documentation is needed for that.

Domain-Specific Language (DSL) : consists of the part that can be formalized, which is the the static semantics and the concrete syntax, and the part, which cannot be formalized, that is the dynamic semantics of the DSL. The notion of semantics is similar to the corresponding notion in the field of linguistics. The semantics of language elements implies their actual meaning. In case of DSL, the semantics of every element in meta-model is its equivalent in the corresponding domain. The semantics of a combination of meta-model elements consequently corresponds to the meaning (i.e. existence, conditions, impacts, value) of the combination of corresponding domain concepts.

Consequently, when using the MDSD approach during the analysis and design phase the main deliverables are:

- DSL's meta-model, containing the information about it's syntax;
- textual documentation explaining the DSL's dynamic semantics;
- model capturing state of the domain's instances.

3.1.2 Model-Driven Development

As was already discussed in section 2.2.3, the software development process resembles the process of mapping from problem to solution spaces.

The MDSD approach as defined in [SVC06] prescribes the following terminology for these constituents:

Platform : Corresponds to the *domain-independent part of a framework*, but is relative to the notion of domain, i.e. depending on which domain is chosen, the domain-independent framework is all the infrastructure, which is lower level, then the implementation of the domain-specific concepts.

A platform can consist of *building blocks*, which correspond to the component and infrastructure libraries as defined in 2.3.2, other frameworks, middleware, plug-ins or aspects in terms of Aspect-Oriented Programming (AOP).

The application code, when using MDSD approach, can be separated into:

Generated Artifact : is the part of the application code that can be generated from from the model according to the DSL's semantics and has its software "module" structure that resembles the domain structure.

Non-generated Artifact : is the manually written part of the application code, which implements application specific logic, that cannot be expressed using DSL; depending on the expressiveness of the defined DSL it can be quite different in it's size.

The mapping from problem to solution in MDSD is done using two mechanisms: the transformations from models to generated artifacts and manual implementation of the software requirements to the non-generated artifacts.

Transformations are the rules of mapping the syntactical elements of the given meta-model to the software solution artifacts or to the syntactical elements of another meta-model. Therefore two types of transformations are distinguished: *model-to-platform (model-to-code)* transformation and *model-to-model* transformations. To be able to define the transformations the special languages are used, they could be considered as the DSL's for writing transformations.

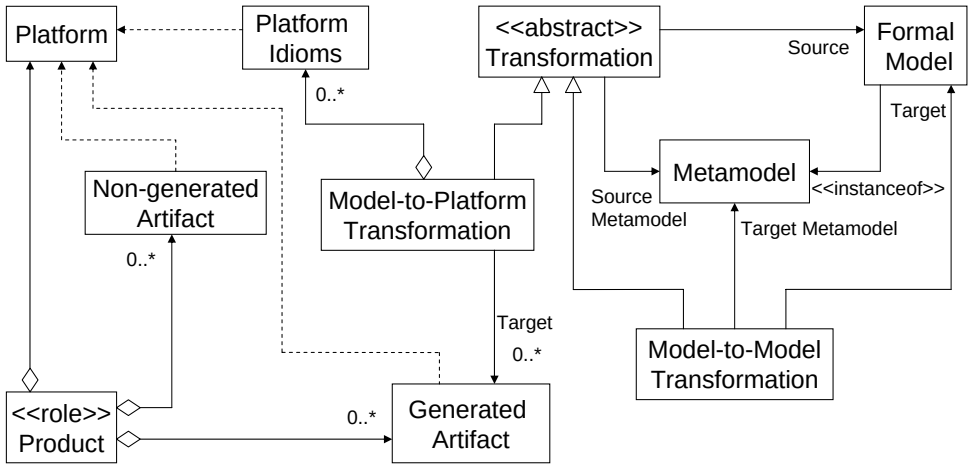

Figure 3.2: Concept formation: transformations [SVC06]

A model-to-model transformation creates another model. However, this model is typically based on a different meta-model than the source model. Such transformations generally describe how the constructs of teh source meta-model are mapped to the constructs of the target meta-model.

A model-to-platform transformation produces generated artifacts, that are based on the platform. For this class of transformation a target meta-model is not needed.

Product : the target working application, which consists of the generated and non-generated artifacts and the platform, which serves as a framework for them.

The relation of all the described concepts is shown in figure 3.2

3.2 Meta-Modeling Explained

Now that the general approach for MDSD is clear, the questions arises: what are the general modeling principles for building DSLs and models and how and why the existing meta-meta-models are capable of having enough expressiveness to be able to define "any" DSL?

3.2.1 Levels of Meta-Modeling

According to the definitions given in the previous section, in the MDSD approach, three levels of modeling can be distinguished:

- M1 - model level;
- M2 - meta-model level;
- M3 - meta-meta-model level.

As shown in figure 3.3 each model is described by its meta-model, each meta-model in its turn is described by its meta-meta-model. I.e. each meta-model is an instance of some meta-meta-model, and each model is an instance of some meta-model.

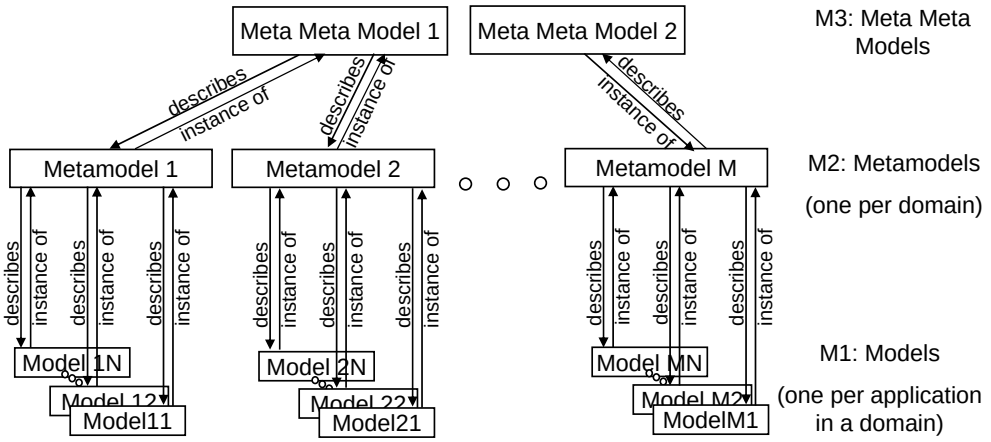

Figure 3.3: Levels of meta modeling

Each meta-model corresponds to one domain and describes models which correspond to applications in this domain. There could be more than one application in one domain. This means, that meta-model is in the one-to-many relationship with the models it defines.

In the same way the meta-meta-model is in the one-to-many relationship with all the meta-models it defines.

3.2.2 Modeling as Formal Real World Description

From the philosophical point of view everything in the real world, that a person can think or speak of, is an *object* (*entity*, or *concept*) [Lay06]. The real world entities are described through their *properties* (*attributes*), *behavior* (which also can be considered as the *behavioral property*) and *relations* to other entities [Swo00]. These philosophical knowledge about the nature of the real world concepts is used in the computer science for the formalization of the real world concepts. The first step on this way was the invention and popularization of the object-oriented programming languages.

The 40-year old history of using object-oriented programing languages proved its convenience for the development of large scale information systems, that have to resemble and support real life human activities. And if at the beginning of the object oriented development the developers were still afraid of using pure object-oriented programming languages like *Smalltalk* and *Java* and to be on the safe side preferred the hybrid *C++*, which supported also structural function oriented approach, nowadays the experience in object-orientation showed that *objects*, *properties* and *methods* provide sufficient expressiveness for most development task.

Currently a broad area of computer science called "ontology" is working on the methodology for the further formalization of the real world concepts and the instruments for this formalization. If the philosophy studies the real world nature as it is, ontology in computer science is trying to formalize this knowledge. It introduces the notions of:

Individuals - the basic or "ground level" objects;

Classes - collections, or types of objects;

Attributes - properties, features, characteristics, or parameters that objects can have and share;

Relationships - ways that objects can be related to one another.

Therefore the information about the real world structure and activities can be conceptualized according to its "functional nature", i.e. the central things we are thinking and speaking of are objects, additional information about the objects, which determine their essence, is reflected through attributes or relationships, if this information associate and relate the objects to other objects. The behavior of objects can be defined as change of the state of objects, the state of object is also described through its attributes.

3.2.3 Relation between models and the meta-meta-model

In the everyday life people give names to different objects: "a pen", "e-mail", "equation", "process", these words are nouns. Human brain knows, that all of them are objects and that they are different objects, because they look and feel differently, i.e. have different properties: "small", "electronic", "mathematical", "changing in time", these words in their turn express different type of information then objects, these words are adjectives. Nouns and adjectives are different parts of speech; there is limited number of them in each natural human language, usually five.

Similar is true not only for natural languages, but also for the DSLs. Whenever the new DSLs are defined they described new special kinds of objects (concepts) by giving them special names, as well as define a set of attributes, describing objects properties, establish the relationships to other objects and, probably, define the operations performing some object specific behavioral activities. Every new DSL's element, which describes objects, actually is an object, i.e. is of type object. Every DSL's element describing properties of objects is an attribute in its sense, i.e. is of type attribute. Similar is true for relationships and operations.

The purpose of the DSL is to hold the information about the concepts used for modeling. It resembles the notion of dictionary for natural human languages. Every word in this dictionary is a noun, adjective, or other part of speech, i.e. it is an instance of some part of speech.

A sentence written in the natural language can be compared with a model. The dictionary of the language in which the sentence is written then corresponds to the DSL. Every word in the written sentence is an instance of the word from the dictionary (has it's syntax and semantics, i.e. meaning). Still every word in this sentence is a definite part of speech, is an instance of noun, or pronoun etc. Similarly any model is an instance of its meta-model (is defined by its meta-model) and also conforms to the corresponding meta-meta-model.

As was said before every modeled object in a model is an instance of its corresponding meta-model concept, and in the most general sense is an instance of object/concept/entity in the corresponding meta-meta-model. Similar for the attributes, relationships, operations.

3.2.4 Ecore Meta-Meta-Model Structure

As was discussed before, the meta-meta-model, the purpose of which is to have enough syntactical expressiveness to define new DSL's has to have concepts (types) to define new: objects, attributes, relationships and operations.

There exist several frameworks supporting the MDSD approach, and each has its own meta-meta-model. Although these meta-models are slightly different, they have

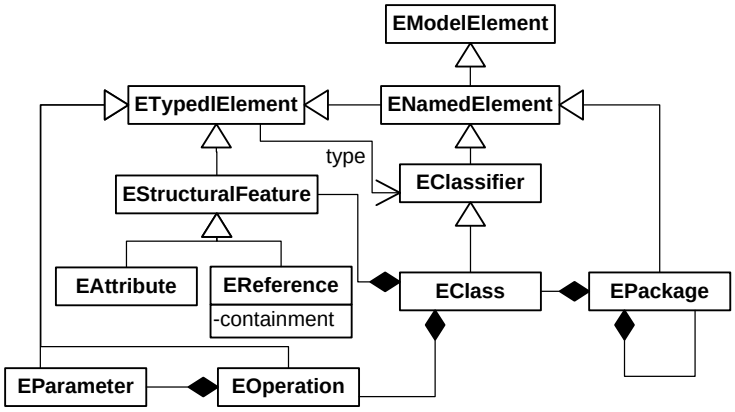

Figure 3.4: Ecore Meta-Meta-Model

similar core structure. Here, as an example the Ecore meta-meta-model is shown, which is used in the Eclipse Modeling Framework (EMF) [EMF]. Simplified the Ecore meta-meta-model is shown in figure 3.4. In more detail it is shown in appendix B.

As shown in figure 3.4 every information element of the developed DSL has to instantiate a subclass of `EModelElement` abstract class. Most modeled DSL elements have names, there fore such elements instantiate a subclass of the abstract class `ENamedElement`.

The concepts modeled with the DSL ("nouns") are modeled as elements of type `EClass`, they can have attributes and references, which describe additional information about the modeled concepts and are instantiating one of the two corresponding subclasses of the abstract class `EStructuralFeature`: Attributes are instances of class `EAttribute`; the relationships between concepts are described through references to other objects, which are instantiating the class `EReference`. The behavioral information about the concepts of modeling is reflected through operations of type `EOperation` and can accept parameters of type `EParameter`. The concepts can be grouped in packages, which instantiate the class `EPackage`.

The classes of type `EClass` define new types (concepts). Other elements like attributes and parameters should themselves be of some type, i.e. have it's type characteristic (e.g. string, integer etc.). But this type is not, the real type which they instantiate (because they instantiate `EAttribute` or `EParameter` correspondingly), therefore the type information is recorded as an attribute of the abstract class `ETypedElement`. In the similar way the information about the returned type by an operation, or the type of the instance to which a reference points is recorded.

The additional information about each of these Ecore *model elements* (i.e. subclasses of `EModelElement` like classes, attributes, references, operations) is described through the attributes of the corresponding *model elements*, relationships and operations. E.g. if a reference to an object implies the aggregation link to this object, then the attribute `containment` of the corresponding reference is set to true.

It is important that the Ecore meta-meta-model is an *instance of itself*. I.e. every Ecore model element is an instance of `EClass`; every attribute of Ecore model element is an instance of `EAttribute`, etc. This means that Ecore meta-meta-model can be considered as a meta-model, the meta-meta-model of which is Ecore meta-meta-model, i.e. it can be seen as one of the meta-models defined using Ecore meta-meta-

model. This provides the possibility to use the same editors, transformations and other tools for Ecore meta-meta-model as for all other meta-models, which is used actively by the EMF tool builders.

Similar is true one metalevel down on the level of models and meta-models. Every meta-model can be considered as a model, the meta-model of which is Ecore. This fact is also used by the tool buiders, and provides the possibility to use the same editors for models (instances of any random meta-model) and meta-models (instances of a definite Ecore meta-model). This fact will play an important role in the current work, for the attaching of the EMF change tracer to meta-models. This will be discussed later in section 5.

3.3 MDSD Tool Support

The described MDSD approach prescribes no assumptions about the technologies that have to implement it. Therefore any technology that implements the described concepts can be considered as the technological solution for MDSD.

The software engineering research and industrial teams during the the past two decades have built numerous frameworks, that aim at implementing MDSD approach or at least some it's parts. Often they are called meta-programming environments, or generative programming environments. The examples of such frameworks developed in the USA are MetaEdit+ [MEd], GME2000 [GME, SLKN01] etc. The modern academical and industrial research effort in the European MDSD community is concentrating around Eclipse Modeling Framework [EMF]. Own separate meta-meta-models are the basis for each of such frameworks.

Eclipse Modeling Framework (EMF) is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model [FB03]. The growing popularity of EMF provides for its fast development as an emerging de-facto industry standard onto which many interesting MDSD tools are built [SVC06]. The core of the EMF framework is the eCore meta-meta-model.

The only currently existing standards in the MDSD field originate from the work of the OMG standardization consortium [OMG], which developed the standards for the MDA approach. Among these standards OMG issued the specification for a meta-meta-model called *The Meta Object Facility (MOF) [MOF06]*. The MOF meta-meta-model is more extended then Ecore meta-meta-model and provide for more expressivness of the meta-modeling. Although several implementations of the MOF meta-meta-model exist, they only implement a relevant practical subset of the MOF. The model repository built within SAP company in MOIN project, discussed in section 2.5, also uses MOF as the meta-meta-model.

Chapter 4

Metamodel Evolution Problem Analysis

4.1 Problem Description

As was mentioned in section 2.4.2 if a change of a DSL is done, the models written using the old version of a DSL may become invalid with respect to the new version of the DSL. In the context of MDSD this means that changes to a meta-model may invalidate the models which are the instances of this meta-model. This problem is the focus the the current work.

Below we illustrate this problem with an example.

4.1.1 Example: Meta-Model Changes

As an example let's define some domain. It can be any area of interest, e.g. business (some enterprise performing some business activity). The domain of interest can be analyzed from the structural or process aspect, i.e. the structure of the company, the structure of the company's environment (customers, suppliers, partners), or its business activities, business processes. Let's assume that we analyze business processes in a company. This is a common domain of interest. A large number of different software consulting companies (including SAP) have to analyze the business processes of their customers and build the corresponding software systems, supporting these processes.

In figure 4.1 an Expense Reimbursement Business Process is depicted using Business Process Modeling Notation (BPMN) (see [BPM06]).

The BPMN is a standardized graphical notation for drawing business processes in a workflow. BPMN was developed by Business Process Management Initiative (BPMI), and is now being maintained by the Object Management Group. The primary goal of BPMN is to provide a standard notation that is readily understandable by all business stakeholders, i.e. the business analysts who create and refine the processes, the technical developers responsible for implementing the processes, and the business managers who monitor and manage the processes.

The example business process depicted in figure 4.1 was taken from [Whi06]. The process is a sample expense reimbursement process. This process provides for reimbursement of expenses incurred by employees for the company. For example buying a technical book, office supplies or software. In a normal day there are several hundreds of instances of this process created.

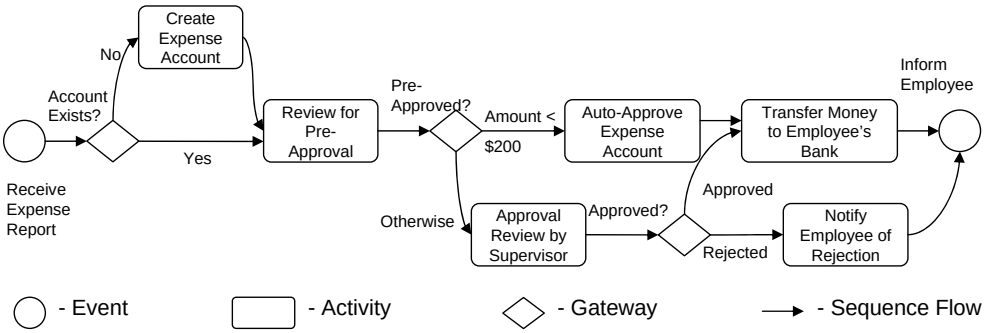

Figure 4.1: Expense Reimbursement Process

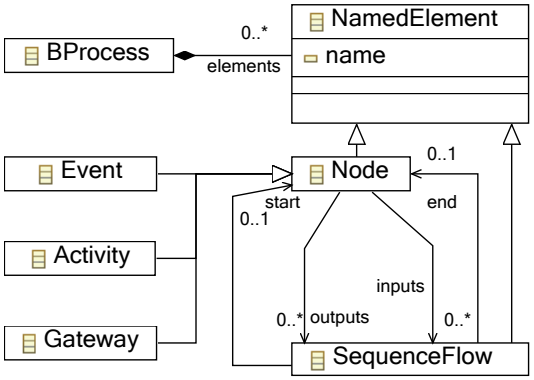

Figure 4.2: BPMN Meta-Model Subset

After the Expense Report is received, a new account must be created if the employee does not already have one. The report is then reviewed for automatic approval:

- Amounts under $200 are automatically approved;
- Amounts equal to or over $200 require approval of the supervisor.

In case of rejection, the employee must receive a rejection notice by email. The reimbursement goes to the employees direct deposit bank account.

In case a DSL for description of processes similar to the described one a meta-model is constructed, which provides elements for capturing and representing the needed aspects. The BPMN diagram from figure 4.1 then can be considered as a model, which is an instance of the meta-model shown in figure 4.2.

This meta-model defines the abstract syntax for the BPMN elements used in the BPMN example diagram from figure 4.1. These are not all the elements defined in BPMN, but a small subset of them. Anyway other business processes besides the depicted one, which can be described only this BPMN subset of elements, can be defined as models instantiating this meta-model.

Non-Breaking Change

But whenever the modeling of other business process elements are needed, besides the ones predefined in the meta-model depicted in figure 3.3, the extension of this meta-

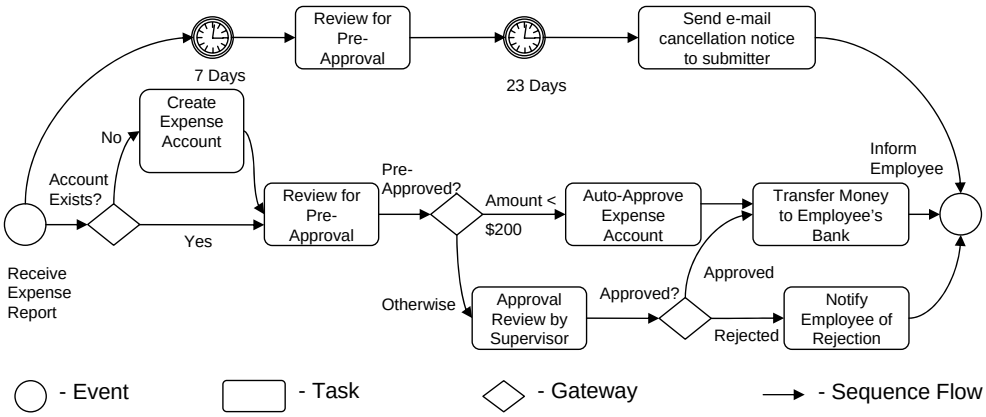

Figure 4.3: Expense Reimbursement Process: Timeout Constraints Added

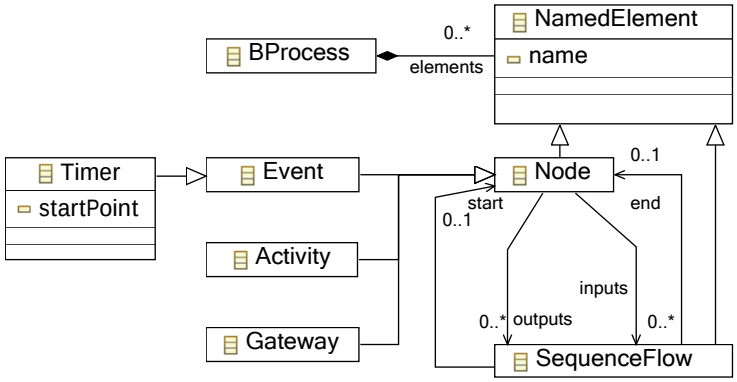

Figure 4.4: BPMN Meta-Model Subset: Addition of Timers

model is needed, e.g. if we want to model the timeout constraints for the expense reimbursement process.

Consider the following timeout requirement for this business process:

- if no action has happened in 7 days, then the employee must receive an approval in progress email;
- if the request is not finished within 30 days, then the process is stopped and the employee receives an email cancellation notice and must re-submit the expense report.

Using BPMN these constraints can be expressed as shown in figure 4.3.

To be able to describe some events that happen at definite points of time BPMN prescribe special kind of events, *timer event*. In context of MDSD, to model such timer events in model we have to add the description (the metaclass) of timers into the corresponding meta-model, as it is done in figure 4.4. Because a timer event is a kind of event, it is added to the meta-model as a subclass of the event metataclass.

Now lets consider which consequences has this addition of the timer meta-class to the meta-model, which describes different business processes.

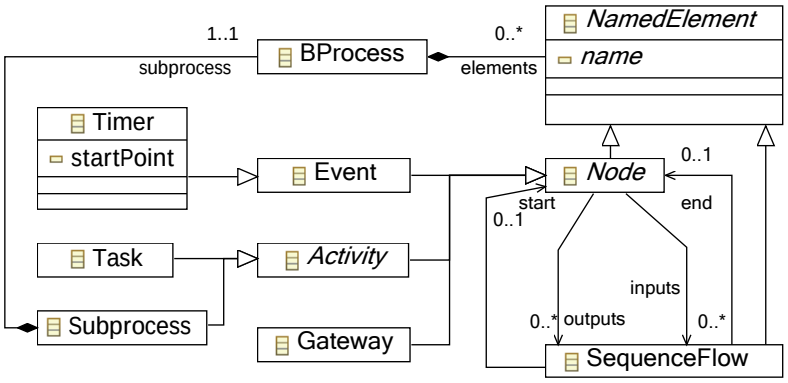

Figure 4.5: Expense Reimbursement Process: Illustrating a Subprocess

Introducing a new subclass to the event meta-class adds new element to the meta-model. This new element gives additional modeling possibilities, i.e. increases the "degree of freedom" in the process of building models. By introducing the new subclass we have not deleted any existing information and haven't changed any existing meta elements. Therefore such change will not invalidate the existing model. The introduced change is additive (non-breaking) or "backward compatible".

If we have several models which were instances of the initial BPMN meta-model shown in figure 4.2, then all these models conform to the old meta-model and consequently do not model any timer events. Therefore the additive change of this old meta-model to the new meta-model shown in firgure 4.4 will not invalidate any models. No migration of model instances is needed, because they remain valid with respect to the new meta-model.

Breaking Change

Another kind of meta-model evolution scenarios is also possible. In case the activity called "Create Expense Account" has to be described in more detail. The creation of expense account may represent a definite subprocess of the expense reimbursement process. To illustrate subprocesses the BPMN uses a special kind of activity. Namely, BPMN prescribes to different kinds of activities: tasks and subprocesses. Tasks are atomic activities or actions, used when the work in the process is not broken down to a finer level of process model detail. On the other hand, subprocesses are non-atomic compound activities, that can be broken down into a finer level of detail and enable hierarchical process development. Every activity in a BPMN diagram has to be of either task or subprocess.

To reflect this BPMN recommendation in out meta-model we have to introduce two subclasses of the class Activity: Task and Subprocess and to define the class Activity as abstract to reflect that it cannot be instantiated and that every activity has to be one of its subclasses. This meta-model modification is shown in figure 4.5. Instances of such meta-model have means to distinguish between atomic and non-atomic activities. In the figure 4.6 an instance of this meta-model is shown.

Now let us analyze the changes that we introduced to our meta-model. Basically, we made three changes:

1. introduced subclass Task;
2. introduced subclass Subprocess;

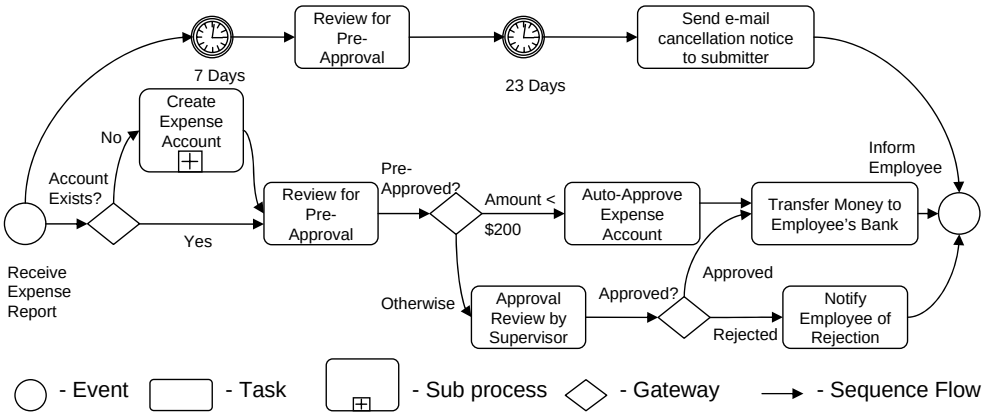

Figure 4.6: BPMN Meta-Model Subset: Addition of Tasks and Subprocesses

3. modified class Activity to become abstract.

Introducing of two new subclasses is similar to the change described before (example with timer addition). Consequently these changes are also additive and non-breaking.

Another case is the third change. Making the class Activity abstract we decreased the "degree of freedom" in the process of model creation. Now the instantiation of class Activity is forbidden. Therefore all the models that contain instances of meta-class Activity become invalid, because they do not conform to the new meta-model, e.g. model shown in firgure 4.1 does not conform to the meta-model from figure 4.5.

Such changes can be referred as breaking. Special efforts have to be undertaken, to continue working with the invalidated models. More about these efforts further in this chapter.

Motivation for the Current Example

The BPMN was chosen for the given example due to its importance in every day software design processes in many software development companies and because its semantics is commonly understood. Another important issue is the absence of a common (standard) working meta-model for this notation.

Although the standardization document exists (see [BPD]), which prescribes the requirements for such a meta-model, no real working meta-model is build up to now. Different software companies understand the importance of tools for business process modeling and have built or are still building them, e.g. SAP has several ones. Still they do not share any common meta-model or have no meta-model at all. Therefore the ability to build models under the condition of absence of stable, commonly accepted and standardized meta-model is of high importance and the instruments for meta-model change management and BPMN model migration are of great importance in this domain.

4.2 Changes Classification

As was shown in previous examples changes of a meta-model may invalidate the corresponding models. By analyzing the meta-models, and their changes, it is possible to foresee the probable impact the meta-model changes may have on to some of the

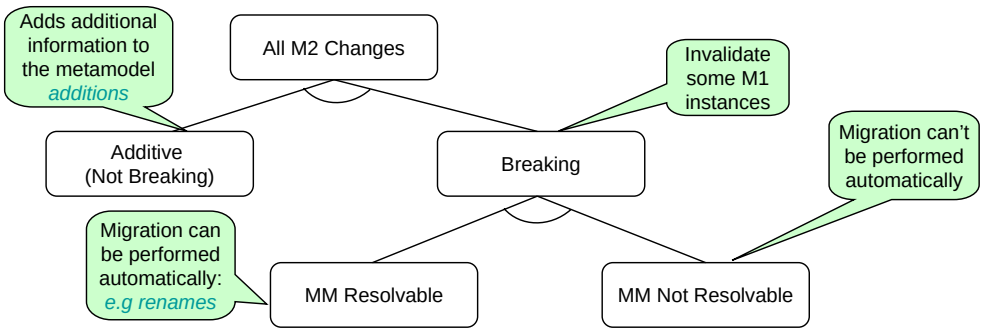

Figure 4.7: Kinds of meta-model Changes According to the Impact on the Corresponding models

corresponding modeles. According to the kind of this impact, we can distinguish the following classes of changes:

Additive (Not Breaking) : These are all kinds of changes that increase modeling "degree of freedom", e.g. all kinds of meta-class additions, additions of attributes to classes (if they are not required attributes), etc.

Breaking : These changes decrease the "degree of freedom" for modeling some elements, they set additional constraints to the models in the modeling domain, or change the existing meta-model entities.

All the breaking changes can be devided in two classes:

Resolvable : Those changes, the impact of which to the existing models can be automatically resolved, i.e. some automatic actions can be executed to modify the models correspondingly to the change of their meta-model, so that it become again valid with respect to the new meta-model version.

Not Resolvable : Those changes the impact of which to the existing models might not be automatically resolved, i.e. some manual actions are needed to modify the models correspondingly to the change of their meta-model, so that it become again valid with respect to the new meta-model version. In case of such meta-model change the manual user effort is needed.

In the figure 4.7 the discussed classification is shown.

The analysis of the meta-models and their changes helps to decide which impact a change may potentially have to the models. Anyway, even breaking changes may not break the existing models. For example, if there are no existing models, then, the meta-model may be changed freely, and no additional effort is needed to adjust the models to the new meta-model.

Often a breaking change invalidates some model instances, but some of them remain valid, because they don't instantiate the changed part of the meta-model, or if the element values of the models fit into the new restricted value range, in case degree of modeling freedom is decreased after change.

Therefore, after the meta-model change analysis is done, the analysis of the model instances has to be performed, to detect the impact the meta-model change has to each separate model. As a result of this analysis the invalidated model instances can be detected. For the invalidated instances either automatic or manual counter actions have to be done, depending on the kind of meta-model change: resolvable or non-resolvable.

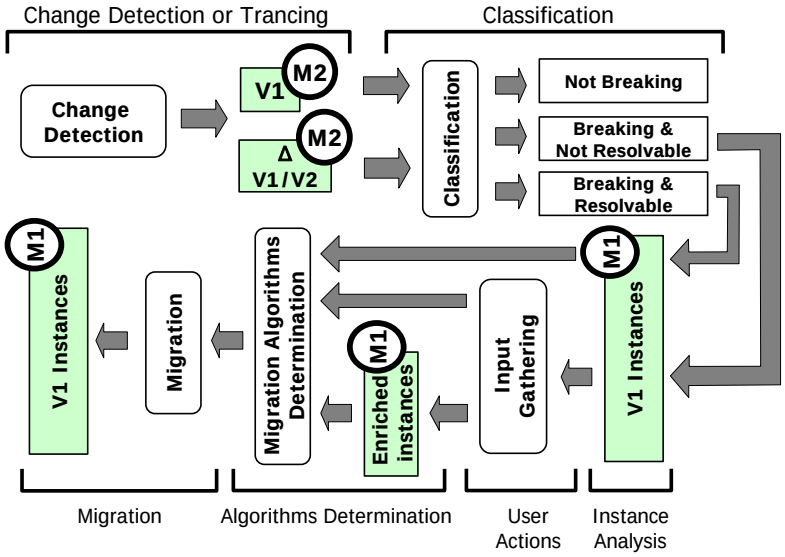

Figure 4.8: Model Migration Process Model

4.3 Proposed General Workflow for Model Migration

The general approach for model migration in case of meta-model change can be split into the following stages:

1. the determination of the differences between the two meta-models;
2. the classification of the changes as described in section 4.2;
3. instance analysis: determination of the invalidated instances;
4. user input gathering: the user input required for non-automatically resolvable changes;
5. algorithms determination: the algorithms required for the migration are determined;
6. the migration is executed.

The workflow diagram of the proposed approach is shown in figure.

The first two stages are the focus of the current work. In the next section the approaches for change determination are discussed. The approach for their classification will be shown in chapter 6.

4.4 Change Detection Approaches

To be able to determine, what kind of meta-model change has happened (for the further analysis of its impact to the corresponding models), we have to have instruments to extract the parts of the meta-model that have been changed (i.e. to be able to separate changed and unchanged parts) and to determine what kind of modifications have happened with these parts, e.g. if they were added, deleted, or modified.

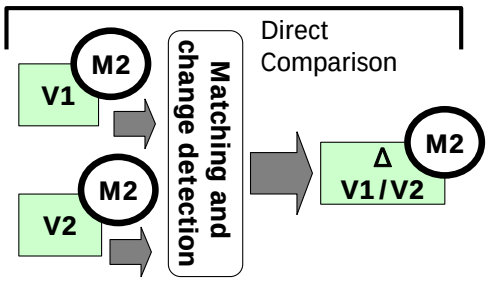

Figure 4.9: Change Detection: Direct Comparison

4.4.1 Direct Comparison

The *direct comparison approach*, in [Gir06] called *offline difference detection* approach (see figure 4.9) is the comparison when two models are compared as independent data structures and no connection between two models a priori exists.

To compare such too models they should be *matched*, i.e. the identical or similar parts have to be found, then the algorithms are executed, which determine the associations between different parts of two models, and further the nature of these differences is determined.

This approach has obvious advantage that two models can be compared whenever the need for that exist. No specific actions are required during the creation of models and their modifications. The only constraint is, that the models should be stored in the same data format, but the can be created and modified with different frameworks and tools.

However the disadvantage of this approach is the computation complexity of the matching/diffing algorithms, their heuristics, that cannot guarantee the detections correct and relevant differences. In case, when the models do not obligatory possess unique identifiers of their elements, the change consisting in the move of some element cannot be unambiguously detected, and may be confused with the action of deletion with the successive addition of a similar element.

The research relevant to this approach is mentions in section 4.5.3.

4.4.2 Change Tracing

The creation of *change traces* (see figure 4.10) requires the modification of the tools used for model creation. A special tracer tool is attached to the model instance and is capable of tracking and collecting information not only about the model modifications but also get the information from the framework, used for model editing. Therefore the tracing approach is more powerful from the point of view of the amount of the information it has at its disposal.

This approach is also called *online* change detection approach (see [Gir06]).

The evident disadvantage of the tracing approach is that only those models can be compared to which the tracer was attached during the modification process. Therefore the tracer tools are desired to be an important part of model editors.

Anyway, this is a minor difficulty, comparing to the advantage that the tracing approach can bring due to the information richness it can collect. Therefore, this approach is the focus of interest in the current work.

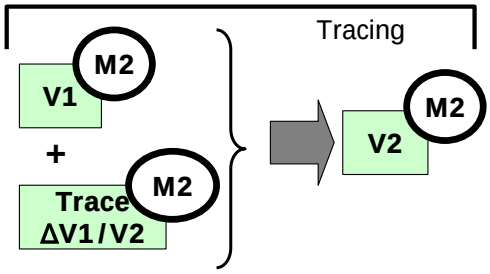

Figure 4.10: Model Migration Process Model

4.5 Related Work

Unfortunately not much research was done up to now in the area of meta-model evolution and the migration of the corresponding models. The current work is based on the approach proposed by Boris Gruschko and is in general described in [BGGK07, Gru07, BG07]. Beside this work the relevant research was done in the some adjacent areas described further.

4.5.1 Domain Evolution. Description of Domain Model Migration

Some effort on the definition of DSL evolution and the migration of the corresponding domain models to follow the DSL changes was done by Jonathan Mark Sprinkle [Spr03, SK04]. In his works Mark Sprinkle refers to DSL changes as a part of the natural process of domain evolution. He distinguishes between two types of domain changes evolutionary changes, which are "small" modifications of some domain parts; and revolutionary changes, "where the paradigm ontologies have few (if any) key concepts in common, and the two syntaxes are more often unrelated than similar"[Spr03]. Here Mark Sprinkle discusses possible patterns of evolutionary changes, i.e. investigates the evolutionary changes that happen with a meta-model during the domain evolution process. In [SK04] Sprinkle defines a domain-specific visual language for domain model evolution, that is a DSL friendly to the concepts of domain model evolution and which provides the user with the ability to create mappings from pieces of the existing meta-model to the evolved meta-model, by assigning the evolutionary pattern to the meta-model elements. Thish language resembles the modern transformation languages for defining the mapping rules of the transformed models, but is more specific, because the domain of its applications is not all possible transformations, but special patterns of evolutionary changes.

The language developed by Sprinkle and integrated into the GME2000 [GME] framework, provides the possibility for the automatic migration of the domain models once the mapping between the two DSL's is defined. The problem of this approach is that the mapping has to be defined manual by the user, even though a convenient language is provided for that; no automatic domain change detection is possible.

4.5.2 Change Classification and Representation

In [AC07] a an approach for model difference (not meta-model!) classification and representation is described, which makes use of EMF framework, Atlas Transformation Language (ATL) [ATL]. In this approach the differences among models conforming

to an arbitrary meta-model can be therefore given as a model which adheres to a difference meta-model obtained by an automated transformation of initial meta-model to difference meta-model. This approach assumes that the difference models are given as firstclass artifacts.

4.5.3 Automatic Difference Detection

A bottle neck of the domain evolution and model migration problem is the development of the suitable algorithm for change detection. There exist a lot of algorithms for comparing general data structures. A overview of such algorithms is given in [TBWK07]. In this work also one generic algorithm is proposed, based on using a high-dimensional search tree for efficiently finding similar model elements in any models irrespectively of the actual format and semantics.

However these generic algorithms are usually week in classifying the changes if they don't do any assumptions about the semantics of the compared data structures, therefore the algorithms adjusted to the specifics of the compared data structures, may be more efficient and are the objects of research, e.g. relational or object databases, XML data, UML models etc.

Up to now no substantial work was done in the direction of the automatic difference detection in meta-models. However, this problem is closely related to the difference detection in UML models. An example of an algorithm suitable for UML models comparison is UMLDiff$_{cld}$ proposed by Gerschick in [Gir06]. A more general approach to the difference creation between two models is given by Alanen and Porres in [AP03], where the mathematical definition of such operations as: difference detection between two models, merging a model with the difference of two models and calculation of the union of two models is given. The approach for conflicts detection of union operation is shown and the way for their automatic or manual resolution. The corresponding algorithms for MOF-based version control system are also provided in this work.

The difference between two models can be detected either by tracing persistent unique identifiers or by matching the similarity, with the further difference calculation (see [TBWK07]).

Model Matching

Matching is the process of detecting identical or similar parts of the compared models. The EMF based tool capable of matching models or meta-models is a part of Atlas Model Weaver (AMW) Project [AMW]. The disadvantage of this tool is common with all similarity based matching algorithms, which is the possibility that after the execution of the heuristic matching algorithm some links between matched elements are still not created correctly, are useless or not created and their manual adjustment is needed. See more in [mat].

Generic Model Management

The general field of *model management* has been defined in [BHJ$^+$00] as the integral process of model matching, merging, difference detection and other model manipulation techniques. The further work of Sergey Melnik [Mel04] discusses the *generic model management* as a common approach for metadata management irrespectively of the application domain.

Tracing

The described difference detection techniques are the techniques for direct comparison of models. The changes trace approach is closely related to the problem of information loss in versioning systems, described by Robbes and Lanza in [RL06].

4.6 Summary

In this section the meta-model evolution problem was illustrated on the example. The example showed that meta-model changes may invalidate the models, which are the instances of the changed meta-model. To solve this problem a comprehensive approach for the detection of meta-model changes, their classification with the further migration of the models onto the new version of the meta modes can solve this problem. The proposition of this approach was done.

According to the proposed migration approach the first two stages of the migration process consist in change detection and classification of the changes performed.

Two basic methods can be suggested for change detection: direct caparison and change tracing. Because the change tracing approach is capable of collecting more information about the performed changes, it promises to be able to deliver more precise results for change classification.

In the current work I will describe how the change classification of the EMF based meta-models can be performed using a standard EMF tracer tool, which is available as part of the EMF framework. I will also give the recommendations about the data that any meta-model change tracer be able to track. These recommendations can be useful for the future development of tracing tools for meta-model change management.

Chapter 5

EMF Tracing Approach For Change Detection and Classification

In the previous chapter the general approach for model migration was described. Change detection and classification are the first two steps of the model migration process. Because the change tracing was defined to be preferable to the direct comparison approach, the current work focuses on this approach.

Building the tracing tools is a complex task. One of the most important requirements for such tools is: what information do such tracers have to track and in which form to store this information? In the current work I am analyzing the types of changes that can be done during meta-model evolution process and define a meta-model that defines the models for storing the information for such changes.

Ideally a change tracer, built from scratch, should be able to store the tracked information as an instance of this meta-model. In the current project, to be able to "test" the designed meta-model, no special tracer is build. A generic built-in tracer tool provided with the Eclipse Modeling Framework is used. The information stored by this tool is later transformed to the instance of the designed meta-model. This information is the object of analysis of the current chapter. The way to use this information for the change classification is also explained in this chapter.

5.1 Tracing in Computer Science

The word "trace" and tracing is one of the widely used words in many sciences. It is used in different meanings in several fields in mathematics, linguistics, computer science and in engineering. Most usages of the term *trace* imply successive, stepwise execution or dependency tracking.

In mathematics and computer science, *trace theory* aims to provide a concrete mathematical underpinning for the study of concurrent computation, including the problems of interleaving and non-deterministic choice. The term "trace" here roughly corresponds to the *dependence graph* [BD96].

Another meaning of the terms "trace" in computer science is: a record of the steps a program has executed. E.g. stack tracing is used for building debugger tools. The sense of tracing in such case is the tracking of the connection between the source code lexemes of the programming language and the real execution of the program,

which enables the stopping of the execution on breakpoints or step-by-step program execution.

In the MDSD area traceability usually refers to the ability of creating relationships between models before and after the transformation [NZR07, Jou05].

5.2 Generic EMF Tracer

The standard EMF tracer is a general purpose EMF facility for supporting/implementing transactions. Originally it was meant to be able to send the changes done to the model by network. Therefore the information saved in traces is very compact and closely corresponds to the way the EMF models are persisted in the XML Metadata Interchange (XMI) documents.

The EMF tracer is capable of storing information of all add/delete/move/change actions that can happen to any models created and edited in the EMF framework. I.e. it is capable of storing this information for any models that are defined by any meta-models, in their turn defined using Ecore meta meta model. Therefore it makes no assumptions about the meta-model, but makes the assumption that the meta meta model of the model is Ecore.

For the task of tracking the model changes the tracer can be attached to a model the meta-model of which is an instance of Ecore meta meta model.

Technologically there is a plugin plugged into EMF+Eclipse framework. If a user wants to save the tracing information, he has to enter the location of the file containing the initial model and start recording. After that the selected model can be modified and the tracing information about the modifications will be tracked. When the modification is finished the collected information can be persisted.

5.2.1 Change meta-model

The information about the model modifications is saved in form of a model, the meta-model of which (simplified) is shown in figure 5.1. The exact version of this meta-model is shown in appendix A. This meta-model is available as a EMF plug-in.

The trace model, as defined by its meta-model, has one container element of type `ChangeDescription`. The `ChangeDescription` element appears once per trace model instance, and is the root element of a trace (when it is persisted in XMI). It holds the information about all the model modifications recorded during one change session. Namely it contains *model elements* added during the change session (containment reference `objectsToAttach`), references to the deleted model elements (`objectsToDetach`), as well as a Map associating the modified model elements with the collections holding the information about the changed structural features of these elements (`objectChanges`). Here added/deleted/modified *model elements* are the objects, whose meta meta type is a subtype of `EClassifier`. [1]

The information about the modified *structural features* (references instances of `EReference` meta meta class or attributes instances of `EAttribute` meta meta class) of a model element is saved as the collection of elements of type `FeatureChange`. This collection, as was already mentioned, is associated with the reference to the modified model element using the Map of type `EObjectToChangesMapEntry`. Each element of type `FeatureChange` from the collection contains the following information: `featureName` (the name of the changed structural feature of the model element), `dataValue` (the new value of the corresponding property, if it can be represented as

[1] Additions, deletions and moves of the model elements are the changes performed in the EMF Tree editor.

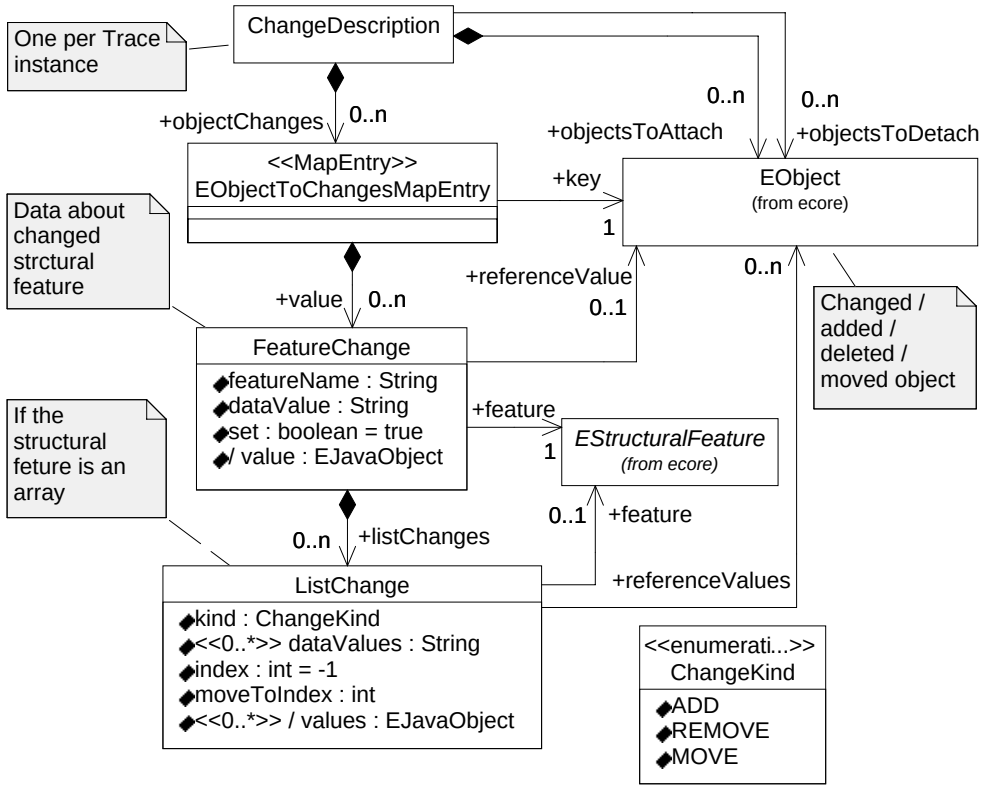

Figure 5.1: EMF Tracer Change meta model

string), `feature` (the reference to the corresponding modified structural feature of a model element). [2]

If a reference in the metaclass defined to have `0..*` multiplicity, this means that in the model instance there could be many instances of a reference with the same name (i.e. a collection). The additions or deletions in the values of these references are reflected as `listChanges` of type `ListChange`.

The instances of `EOperation` meta meta class are not parts of the model, they are expressed only the source code. [3] Therefore the only data saved in the trace refers to the additions/deletions/moves of model elements [4] and changes of structural features of these model elements. [5]

Basically the structure of this meta-model is designed to be aware of the structure of the XMI documents only.

5.3 Ecore Semantics: Influence to the M1 models

As was already said the EMF tracer can be attached to any model the meta meta model of which is Ecore. Therefore the changes are detected and represented based

[2] These are all the changes that can be done in the EMF property editor.
[3] EMF has no editor for modifying operations.
[4] Performed in EMF Tree editor.
[5] Performed in the property editor.

on the semantics of the meta meta model of the edited model, i.e. Ecore semantics. No assumptions about the semantics of of the meta-model is done.

Basically, the Ecore meta meta model prescribes expressing the models through instances of classes, and describing additional properties of these instances through attributes and references. Therefore the following semantical elements of Ecore meta meta model can be distinguished:

- *model elements*, instances of `EClass` meta meta class and persisted as XMI elements (resemble the ontological notion of "concepts" or "entities");
- *attributes*, instances of `EAttribute` meta meta class and persisted as XMI attributes (resemble the ontological notion of "attributes" or "properties");
- *references*, instances of `EReference` meta meta class and persisted also as XMI attributes, but with the semantical constraint that only existing object can be referenced (resemble the ontological notion of "relationships" among "entities");
- *containment references* to model elements; the referenced model elements are persisted in XMI as child elements, but with the semantical constraint that only existing object can be referenced (resemble the ontological notion of "part-of relationships");
- *generalization relationships* among model elements, semantically basically means that some classes have common sets of structural features (references or attributes), is not directly reflected in XMI, but through duplication of the structural features in all the subclasses of a common super class (resemble the ontological notion of "subsumption relationships");

In the next sections the explanation of the EMF tracer mechanisms for dealing with these semantical constraints is described.

5.3.1 EMF Tracer Semantical Constraints: Containments

One of the assumptions is done about the containment references of a model element. If a M1 model element (an instance of `EClass` meta meta class) has contained model elements (which in M2 level is modeled through containment reference to other model instances of M3 `EClass`), then the deletion of this M1 model element will cause the deletion of the contained model elements. This is done, because on the M3 level `EClass` has reference called `eAllContainments`. Therefore, when a model element of `EClass` meta meta class is deleted, the EMF knows that the meta class of this element has one to many reference (instance of `EReference`) called `eAllContainments`, that references instances of another meta-meta-class. The EMF reads the array of type `EReference` called `eAllContainments` of the M2 object (of type `EClass`) and gets the value of the `EType` reference of each such reference in this array, the M2 objects of M3 `EClass` referenced by `EType` are then correspondingly deleted.

The impossible existence of the model elements without their container elements is the central semantical rule of the Ecore meta meta model. The consequences of it is the constraint are commonly understood for M2 models, e.g. classes have to exist inside a package, or that it is impossible to create an attribute which is not contained in any class. Basically these constraints are modeled through containment reference between package and its classes, classes and its structural features in the Ecore meta meta model.

The deletion of an object with all the contained sub-objects is considered as one change, due to the assumption that an object cant exist outside its container.

Meaning for the XMI Document

From the point of view of the persistent representation of models, the containment reference basically means that the element is contained inside another container element. Therefore if the container element is deleted, everything inside it is deleted too.

5.3.2 EMF Tracer Semantical Constraints: Referenced Objects

Another important semantical constraint is that references cannot reference the deleted objects. Therefore when an object is deleted, it sends the event that it was deleted. As the result the value of the reference to this object in other objects is set to `null` (i.e. unset).

In the case of deletion an object and the corresponding references to it, the two deletions are accepted as different changes. The use of the eventing mechanism does not give the possibility to distinguish between two events, if the reference should be deleted because of the deletion of the referenced object, or because of the deletion of acually the reference itself by the user in the property editor.

Meaning for the XMI Document

Because in the XMI document an object and a reference to it are represented as different independent XMI elements, the changes to both of them are also considered as independent.

5.3.3 EMF Tracer Semantical Constraints: One-to-many References

The one-to-many references referring the collections of model elements in the XMI are shown as indexed sequences of values (or namely IDs of the XMI elements they reference). This is a pure consequence of the way, the models are persisted in XMI. If the persistence format is different, then this would not be an issue, the one-to-many references would be persisted as one-to-one references, and their changes would be reflected in the same way.

5.4 Tracing the meta-model Changes with the EMF Tracer

In the previous section the work of EMF tracer is introduced and the meta-model describing the structure of the traced changes was explained. As was already mentioned this tracer can be attached to any model, the meta meta model of which is Ecore. Consequently it can be attached to a meta-model (M2), the meta-model of which (M2+1 = M3) is Ecore, and the meta meta model of (M2+2 = M4, i.e. the meta-model of Ecore meta meta model) is again in its turn Ecore. The relation among these meta-modeling levels can be also considered as:

- M2 meta-model are “living” on the level of M1 models (M1’ = M2);
- Ecore is “living” on the level of M2 meta-models (M2’ = M3);
- Ecore is ‘living” on the level of M3 meta meta model (M3’ = M4).

The tracer treats then the meta-model as a model (M1' = M2) based on its Ecore meta meta model(M3' = M4) and has no knowledge about the meta-model of the traced instance (M2' = M3). This meta-model in case of attaching the EMF tracer to any M2 instance is Ecore.

5.4.1 Trace Refinement Problem Description

In the previous section this mechanism for tracing changes in any models based on random meta-model is explained. The definite advantage of the EMF tracer is the capability of detecting changes for any models, making the assumption only about its meta meta model.

For the task of model migration, the object of the current research, it is of a great importance to have the explicit information about the change dependencies. Because each change is classified as additive/resolvable/non-resolvable for the further assigning of the appropriate migration algorithm, it is important to have the explicit information about the "transactional" dependencies of the changes. The type of the a composite change strongly depends on the types of it's sub changes, and therefore, if one of sub changes is not resolvable, then the whole composite change is non-resolvable too. E.g. if a model element is deleted then its type (resolvable/non-resolvable) can be assigned only if the types of deletions of all its contained sub-elements is explicitly analyzed.

It is also important to know the semantics of the Ecore elements. I.e. what exactly each change of each element of M2 models means, because each M2 model element or structural feature poses definite meaning to the corresponding M1 models and is important for the M1 model migration process.

The general approach to be able to detect composite changes is to enrich the traced information with explicit change dependence information. In the context of the MDSD approach such operation implies a transformation of the trace instance and the meta-model instance to the model, holding the information about the change dependencies, which is an instance of a meta-model specially designed for holding such information. The next chapter describes the requirements for such target meta-model and its design.

Chapter 6

Trace Refinement

6.1 Requirements to the meta-model For Change Classification

Because each element used for creation of M2 models is defined in Ecore M3 model (common DSL for M2 model creation), it has a commonly understood meaning (semantics). When building the M2 models using these M3 modeling elements and features semantical meaning (semantical constraints) are defined for the M2 models (the syntax and semantics of the DSL). This semantics influences the corresponding M1 models, defining the form they have to conform to be valid M1 models with respect to M2. Consequently, if the M2 models change, the changes may invalidate the corresponding M1 models, and the nature of this invalidation depends on what modeling element (feature) was changed in M2. Therefore, to classify all possible changes that can happen to M2 models and their influence to the corresponding M1 models we have to analyze the following things:

- the modeling element (feature) than was changed in M2 model (its semantical meaning defined by Ecore M3);
- the decencies that these changes have to the M2 model (by analyzing the the structure of M2 instance);
- the instances of M1 models that conform to the change M2 model (the values of the changed M2 elements and features).

Therefore for the task of creation of a Ecore sematnics-aware refined traces, it is desired that every type of a change corresponds (is associated) with the corresponding Ecore model element. Therefore the new meta-model for change classification should resemble the Ecore M3 structure and semantics.

Another requirement is that the meta-model should have the appropriate design to be able to store the information about change dependencies as was described in the previous chapter.

The last nice-to-have requirement is the possibility to combine traces. I.e. if there are three versions of meta-models M2, M2' and M2" and the traces between M2'-M2 and M2"-M2' the possibility to detect changes between M2"-M2 is desired.

6.2 Semantics of Ecore Modeling Concepts

Every M2 model element, which is an instance of Ecore `EModelElement` subclass in the process of modification can be added, deleted or moved. The consequences of such changes to the corresponding M1 models can be additive, resolvable or non-resolvable as shown in the table D.

The additional information about the M2 modeling element, defining their properties, is expressed through setting the corresponding values of the model element's structural features. The changes to these values also may invalidate the M1 models. The analysis of such changes is shown in the table D.2.

6.3 Composite Changes Analysis

By analyzing the Ecore meta meta model, the conclusion about the change dependencies can be done. The dependencies about changes can be expressed in form of changes and their sub changes, that consequences of their super change. In the following sections all such dependencies are listed.

6.3.1 Additions

By the addition of a model element all its required structural features must be set. Actually in the Ecore model quite a few structural features of model elements are marked as required. This is done, because the M2 models may exist (be edited or persisted) without these features set, but the instantiation of the corresponding M1 models is impossible. Therefore by the word "required" we actually mean the necessity for the creation of valid M1 instances of the model element.

Additions of Packages: PackageAdd

Subchanges:

- Rename - name should be set;
- NSURIChange - nsURI should be set;
- NSPrefixChange - nsPrefix should be set.

Additions of Classes: ClassAdd

Subchanges:

- Rename - name should be set.

Additions of Typed Elements: ReferenceAdd, AttributeAdd, OperationAdd, ParameterAdd

Subchanges:

- Rename - name should be set;
- TypeChange - eType should be set.

6.3.2 Deletions

Deletions of M2 model elements require the cascade deletion of the contained model elements in M2 models.

The deletion of a model element will also unset the `eType` structural feature, where the value of `eType` is set to the deleted model element. Basically, `eType` value is the instantiation of `eReference` between an instance of `eReference` model element and an instance of `eClassifier`. The unset of the `eType` will invalidate the corresponding `eReference` instance untill some actions, e.g. its deletion or move (change of `eType`) will be done. This invalidation means that the M1 instances of the these M2 `eReference` instances are no more valid and have to be deleted. Therefore the deletion of a model element on M2 level will cause the deletion of all the objects instantiating this model elements. Their deletion will cause the deletion of all references to these objects from other M1 objects.

Consequently, two types of subchnges changes can be detected:

- M2 level subchanges, when the effect of a change to M2 models is visible and analyzed;
- M1 level subchanges, when the effect of a change to the corresponding M1 models is predicted; to consider these possible subb

The first group of subchanges is the consequences of the compositional integrity of M2 models as defined in Ecore M3. The second group of subchanges are the actions preserving the referential integrity for M1 models as defined in M2 level.

Deletion of Packages: PackageDelete

Package is not an instance, but a way of grouping M2 model elements. Therefore it cannot be directly instantiated on M1 level and therefore have no M1 level subchanges.

Subchanges:

- PackageDelete - deletion of contained sub-packages.
- ClassDelete - deletion of contained classes.

Deletion of Classes: ClassDelete

Subchanges (M2 level):

- AttributeDelete - deletion of contained attributes;
- ReferenceDelete - deletion of contained (outgoing) references;
- OperationDelete - deletion of contained operation;

Subchanges (M1 level):

- ReferenceDelete - deletion of incoming references, which in the M2 level means the invalidation of model elements of type `EReference` and on the M1 level means the real deletion of references to the deleted class instance.
- Deletion of the instances of the deleted M2 Class means that the part of the class that is defined in the superclass is also deleted, therefore the structural features contained in the superclass and incoming to the superclass references have to be analyzed and their instances deleted.

Deletion of References: ReferenceDelete

Subchanges:

- OppositeChange - if the eOpposite reference is set, then the eOpposite structural feature of the opposite reference have to be unset. This is a consequence of the deletion of a model element of type `EReference` and the deletion of a reference to this model element, which is in Ecore M3 defined as a reference to a model element of type `EReference` and called `eOpposite`.

Deletion of Operations: OperationDelete

Subchanges:

- ParameterDelete - deletion of contained parameters.

Deletion of a Superclass

If a superclass is deleted from an M2 model, this is an ambiguous change. The static semantics of Ecore meta-model is not sufficient is this case, the dynamic semantics of the DSL has to be considered also. The deletion of a superclass semantically may mean the that its structural features should be moved to his subclasses or that the subclasses (or some of them) do not need the features defined in the superclass any more. Therefore it is an example of automatically non-resolvable change and the decision about the consequences of this change should be defined by the user.

6.3.3 Changes of StructuralFeatures

Change of the eSuperTypes of a class: SuperTypeChange

Semantically for M1 models it means addition/deletion of a set of structural features to a class.

Change of the eType of a reference: TypeChange

Semantically for M1 models it means addition/deletion/move of a reference.

Change of the eOpposite of a reference: OppositeChange

Semantically it means the same action for the opposite reference.
Subchanges:

- OppositeChange - the corresponding change of eOpposite in the opposite reference.

6.4 Design of the Refined meta-model

6.4.1 Composite Changes

To store the dependencies among the changes the mechanism of change containment can be successfully applied. The parent change than causes subchanges would contain these subchanges as children, as shown in figure 6.1.

6.4.2 Change Classification Structure

As defined in section 6.1 the meta model for change representation has to define different classes of changes and associate them with the corresponding Ecore model element or feature.

As was already discussed each model element can be added/deleted/moved or its structural feature can be changed. As shown in appendix D the resolvable/non-resolvable property of additions, deletions or moves of different model elements depends on the types of these model elements, i.e. to decide if addition/deletion/move of a model element is resolvable or not the information about the type of this element is needed, as well as the analysis of the corresponding structural features of this element. Therefore, because the nature of this analysis depends on the type of the model element and its structural feature, it is reasonable to classify the changes not only according to the action performed (add/delete/move) but also according to the types of elements. Consequently the following classification, shown in figure 6.1 can be proposed.

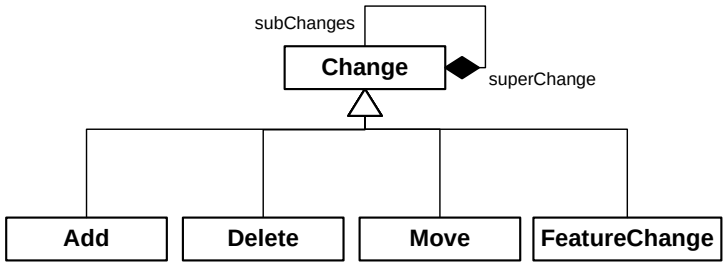

Figure 6.1: Main Change Supertypes

The structural features of model elements can also be changed. The kind of these changes depends on the type of the structural feature. The decision if such change is resolvable or not depends on the structural feature and on the model element containing it as shown in table D.2. The changes of the structural features can be classified by the types of the these features as shown in appendix E

6.4.3 Association Classes

Each change class from the structure shown in appendix E has to have a reference to the corresponding model element or its structural feature that was changed. To fulfill the requirement of trace combinations the meta-model for change classification has to provide the possibility for associating exactly one change class with each model element or its structural feature. Therefore the one to one bidirectional association has to associate each change class with the corresponding model element or its structural feature.

Because the EMF semantics does not have the means to define the bidirectional associations directly, but only through a pair of opposite references, to associate one change class with one Ecore model element a change to the Ecore meta-model would be needed. Another approach is to use the association classes. I.e. the classes that repeat the Ecore structure and reference its model elements and structural features with one-to-one unidirectional reference and similarly have one-to-one unidirectional references to the corresponding change classes. The example of such association for the M2 model elements instantiating Ecore `EClass` is shown in figure 6.2.

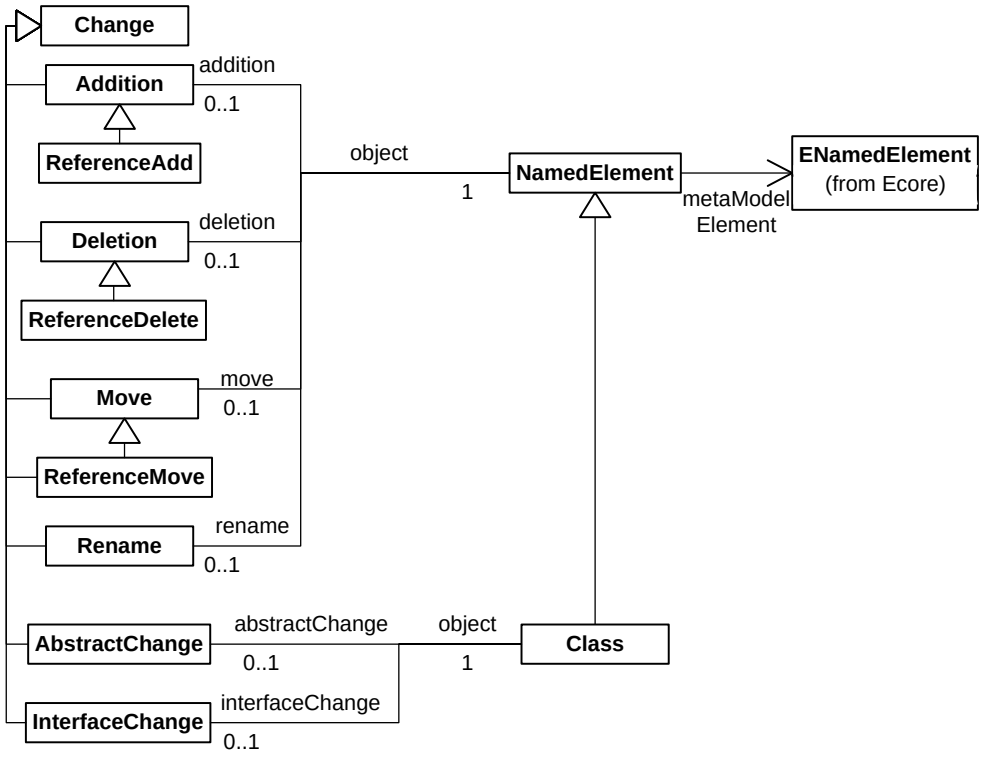

Figure 6.2: Example: Classification of the Class Changes

The instances of the association classes resemble the structure of the M2 models.

6.4.4 Association Classes Structure

The association classes repeat the Ecore class structure. The difference in the structure of the association classes with the Ecore model is the addition of one more reference from the referenced model elements to the references refferencing them, i.e. the structure of the association classes resemble the structure of the M2 models but all the references here are bidirectional. This is done to provide easier detection of reference deletion, when the referenced model element is deleted.

The structure of the association classes is shown in figure 6.3.

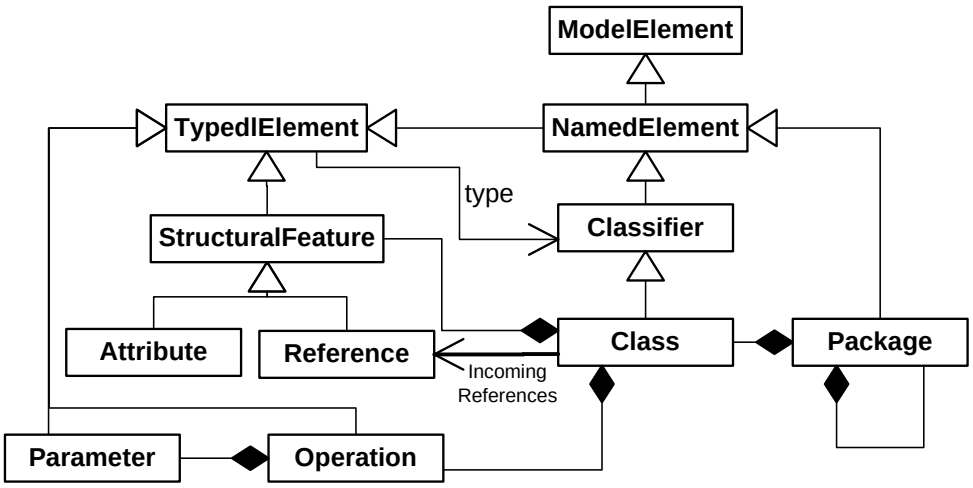

Figure 6.3: Association Class Heierarchy

Chapter 7

Transforming Change Meta-Model to the Refined Meta-Model

Transformations are the main mechanism by which the models are transformed to other models with different meta-model. To build transformation different transformation languages exist. The syntax and semantics of some of these languages is also defined in form of a meta-model (e.g. ATL), some languages do not have it. The overview of the transformation languages and their features is described in [CH03].

The Atlas Transforamtion Language (ATL) is a popular transformation language for EMF based model-to-models transformations, is conveniently integrated in the EMF framework and possessed extensive documentation, therefore the change trance refinement transformation described in this work was done in this language.

7.1 Transformation Configuration

As was said before the trace saved by EMF tracer is not aware of the meta-model that was the object of change. To be able to analyze the change dependencies the knowledge about the meta-model structure is needed. Therefore, to produce an instance of the meta-model for change classification described in the chapter 6 the transformation has to have two input models: the trace and the instance of the original meta-model, that has been modified. This approach has been prototypically implemented and tested. It is schematically depicted in figure 7.1.

The problem with this approach is the inability to accumulate changes from several trace instances. To be able to fulfill this requirement the two-step transformation can be suggested:

Initializing Transformation - the first transformation takes the M2 meta-model as its input and produces the empty meta-model for change classification (`ChangeMM`), which consists only of the association classes part, referencing all the elements of the meta-model under change. No change classes are instantiated. The produced output corresponds to the situation, when no changes were done to the meta-model.

Refining Transformation - the next transformations refine the output of the previous one (initializing or previous refining transformation), by taking the `ChangeMM`

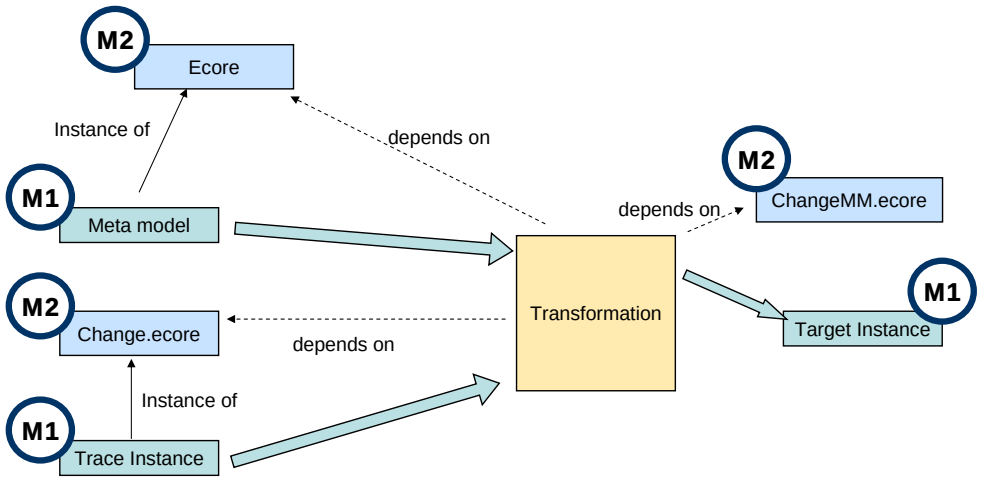

Figure 7.1: General Transformation Scheme

meta-model and the trace instance as inputs and producing the enriched version of `ChangeMM` instance instantiating the change classes in correspondance to the changes detected in the trace and attaching them to the corresponding association classes.

This approach is shown in figure 7.2. With this approach the possibility exist to analyze sequences of traces and accumulate changes recorded during several tracing sessions. In the further sections this approach will be described in more details.

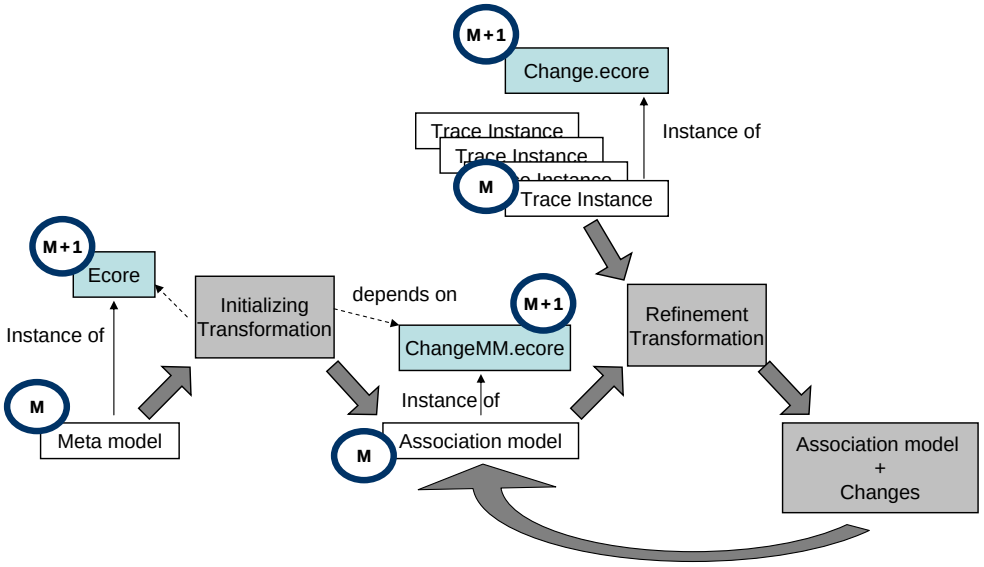

Figure 7.2: Two-Phase Transformation Scheme

7.2 Initializing Transformation

This transformation is the a simple decoration transformation. The input of the transformation is an M2 model. Each class in this M2 model (i.e. instance of `EClass`) should be associated with the corresponding association class from the association class hierarchy shown in figure 6.3.

To produce the association classes structure as an output of the transformation each model element of M2 model has to be mapped to the corresponding ChangeMM association class. In ATL it is done in the following way:

```
module initTransformation;
create OUT : Changemm from MODEL : EcoreMM;

rule EcoreNamedElement {
  from
    ine : EcoreMM!ENamedElement
  to
    one : Changemm!NamedElement (
      xmiID <- ine.__xmiID__
    )
}

rule EcorePackage extends EcoreNamedElement{
  from
    ip : EcoreMM!EPackage
  to
    op : Changemm!Package (
      elements <- ip.eClassifiers->union(ip.eSubpackages)
    )
}

-- etc. for every model element from the association classes hierarchy
```

The problem of ATL transformation language is that it is technically inconvenient to produce the output model that can hold links to its input model. Because each input model element is transformed (mapped) to a new output element, the attempt to reference the input model element from the output model element results in creation of a reference to transformed input element. The proper way of establishing links from the output model elements to the corresponding input model elements in ATL is described in [Jou05].

Another work -around is to store the initial IDs of model elements in the output model of initializing transformation, and with the help of auxiliary refining transformation (which uses this model with stored IDs as its input) to produce a model enriched with the references to the elements of a initial model by matching the IDs of the elements. This transformation is shown in the following listing.

```
module RefiningSubTransformation;
create OUT : Changemm refining MODEL : EcoreMM, IN : Changemm;

helper

  def : elements : EcoreMM!ENamedElement =
    EcoreMM!ENamedElement.allInstancesFrom('MODEL')->asSequence();

rule meta-model {
```

```
  from
    ine : Changemm!meta_model
  to
    one : Changemm!meta_model (
      elements <- ine.elements
    )
}

rule NamedElement {
  from
    ine : Changemm!NamedElement
  to
    one : Changemm!NamedElement (
      xmiID <- ine.xmiID,
      meta_modelElement <- thisModule.elements
          ->select(i | i.__xmiID__ = ine.xmiID)->first()
    )
}
```

7.3 Refining Transformation

The refining transformation has to be capable to read the elements of a meta-model and check if they can be matched to any rule. The condition for matching is met if the set of changes from the change model contains the information about the changes of this element.

In the listing below part of this refining transformation is shown, which is matched in case an element of M2 is renamed.

```
module RefiningTransformation;
create OUT : Changemm refining IN : Changemm, TRACE : Change;

helper

  def : changeDescription : Change!ChangeDescription =
    Change!ChangeDescription.allInstancesFrom('TRACE')->asSequence()
        ->first();

helper context Change!ChangeDescription

  def : getElementsToRename() :
    Sequence (Change!EObjectToChangesMapEntry) = self.objectChanges
        ->asSequence()->select(f |
        (f.key.oclIsKindOf(Ecore!ENamedElement)) and
        f.value->exists(k | k.featureName = 'name'))->asSequence();

helper context Changemm!NamedElement

  def : isRenamed() : Boolean =
    thisModule.changeDescription.getElementsToRename()->exists(i |
        i.key = self.metamodelElement);

helper context Changemm!NamedElement

  def : getNewName() : String =
```

```
    thisModule.changeDescription.getElementsToRename()->select(i |
        i.key = self.metamodelElement)->last().value->select(k |
        k.featureName = 'name')->last().value;

rule EcoreNamedElement {
  from
    ine : Changemm!NamedElement (ine.isRenamed())
  to
    one : Changemm!NamedElement (
      xmiID <- ine.__xmiID__,
      rename <- if (ine.isRenamed()) then thisModule.Rename(ine)
          else OclUndefined endif
    )
}

lazy rule Rename{
  from
    ine : Changemm!NamedElement
  to
    one : Changemm!Rename(
        newValue <- ine.getNewName(),
        object <- ine,
        superChange <- one.object.addition
    )
}
```

7.3.1 Scalability with Meta-Model Size

This implementation approach does not scale good with model size, because every element has to be reviewed independently from the amount of changes made. For big models this results in exhaustive search of changes in unchanged model regions. An optimization of this approach is feasible if the model is partitioned and only changed model partitions are checked (as e.g. marked "dirty").

However, currently no transformation languages provide efficient mechanism for the refinement transformations. The existing transformation languages either not support refining transformations, or have no efficient algorithms for performing optimization on big models when only a small part of a model is refined by the transformation. Conceptually the efficient performing of such operations is an important purpose and challenge of the model refinement, which is a key issue of MDSD.

Chapter 8

Conclusions and Future Work

8.1 Summary

Within the scope of the current Master Thesis an overview of the existing software engineering approaches for the solution of software evolution and complexity problem was done. The trend of the modern software engineering is the shift from program code development to the modeling of the application domain. A promising approach is in this area is the Model-Driven Software Development, which considers a domain model and program code as two facets of the produced software. Therefore the problem of software evolution in MDSD closely relates to the problems of domain evolution and model evolution.

The semantics of domain is formalized in form of a meta-model, therefore, the domain evolution problem equals to the meta-model evolution problem. A particular part of the real world is then described using the meta-model elements and stored in form of a model. The existing related work in this area was studied and the problem of models migration when their meta-model is changed was illustrated. The general workflow for model migration was studied.

The first stage of any model migration process is the detection and classification of changes for the estimation of their severity and choose of the appropriate migration scheme. In the current work two possible approaches for change detection were described and evaluated: the direct comparison and change tracing. The tracing approach was decided to be more flexible and powerful (its power only on the design of the tracer), because it is capable of collecting and analyzing more information about how the changes to the meta-model were done, thus providing the possibility for the unambiguous change detection, which is an irresolvable problem of the direct comparison approach.

In the current work the design of a meta-model for change classification was proposed. The traced changes should be accumulated in the form of an instance of this meta-model. In the work it was explained that to be able to classify changes the semantics of the M3 meta-model of the changed M2 model has to be analyzed, to detect the composite change dependencies, because these changes have to be accepted or rejected as one transaction.

The model containing the classified changes can be obtained from the specially build tracer, that can react to the change events in the meta-model, analyze their semantics, classify and save them in form of an instance of this meta-model (online change classification) or the semantics of the meta-model of the traced model can be analyzed later, when a session of changes is finished (offline approach). With the second approach the same tracer can be attached to the models with different

meta-models. The traced results are later enriched with the meta-model semantics of the traced instance. In the case when the M2 models are traced their meta-model is always Ecore M3. This approach was the focus of the current work, due to availability of such general purpose tracer in the Eclipse Modeling Framework.

For the enrichment of the instances traced by the standard EMF tracer the transformation adding the semantics of Ecore M3 was developed. The advantages of two-phase transformation were discussed, such as the capability of accumulating changes collected from several successive traces.

8.2 Results and Conclusions

The main results of the given Master Thesis are following:

- The design of a meta-model for change classification is build, which is aware of the semantical constraints of Ecore M3. Although it is not the only possible design variant for change classification meta-model, the main requirements to this meta-model were clarified, and the proposed solution fulfills these requirements.
- The transformation enriching the standard EMF tracer model with the Ecore M3 semantical constraints is developed. The implemented transformation is capable of storing the information about the change dependencies, types of changes and of associating changes with the corresponding M2 model elements. The disadvantage of it is the impossibility of change accumulation from several successive traces.
- A two-phase classification approach was proposed consisting in a transformation for initialization of association classes decorating the Ecore M3 and the refining transformation for associating the traced and classified changes with the corresponding association classes. This approach is capable of change accumulation.
- The transformation of change traces to the designed meta-model for change classification faces the problem of scaling with the size of M2 models. However, it is considered to be the problem of tools for definition of refinement transformations. If this problem is a key practical constraint the functionality of this refining transformation can be implemented also using some imperative programming language, e.g. Java.

8.3 Future Work

Currently the following directions for the future work can be foreseen:

- It should be further analyzed exactly which information is needed to be stored in change classes of the developed meta-model. This information should be sufficient for building a transformation that takes the the model with classified changes as its input and for each type of input change-class: the M1 instances, and prescribes the rules of how the M1 instances have to be changed to become valid with respect to the new M2 model.
- The approach for the semi-automatic resolution of non-resolvable changes should be further studied. Information and in what form does the user have to see to be able to resolve the changes manually, in what form input gathered from the user has to be stored. A possible way is to store the user input in the corresponding changed class of the `Changemm` meta-model designed in this work.

- Another possible way to build the change classification meta-model is not to use association classes to associate changes with the corresponding M2, but to build the decorator model of the M2 model elements. The implementation of the decorator pattern for Ecore models should be studied.

Appendix A

Change Metamodel Complete

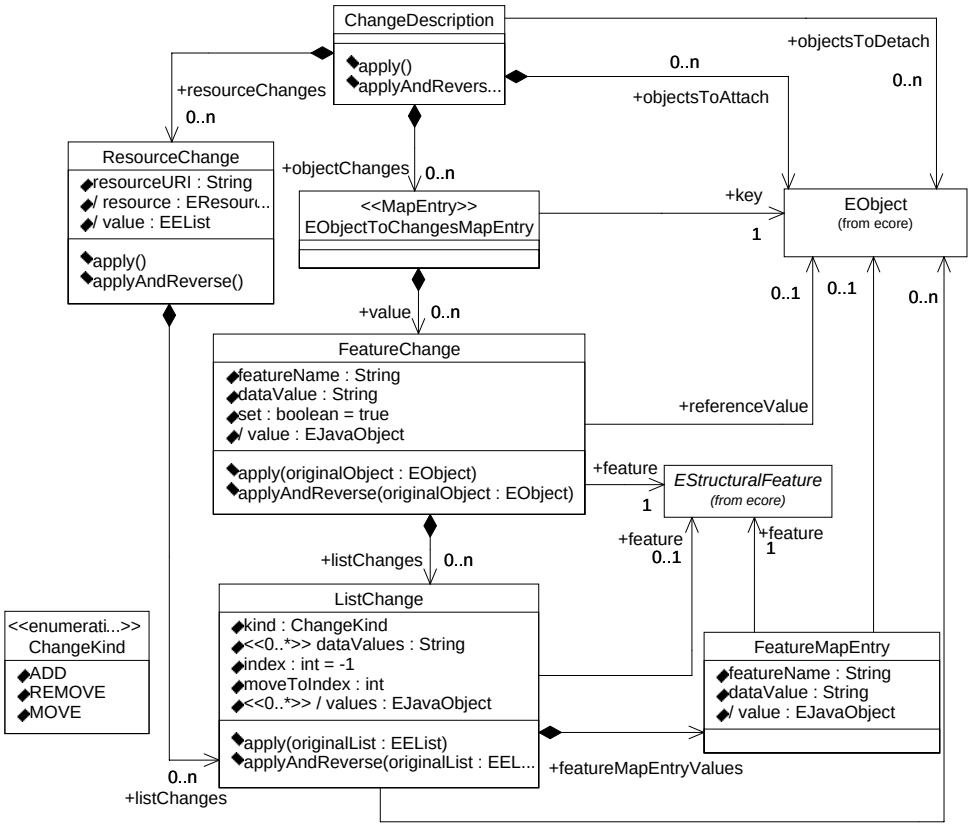

Figure A.1: Change Meta-Model

Appendix B

Ecore Meta-Meta-Model Complete

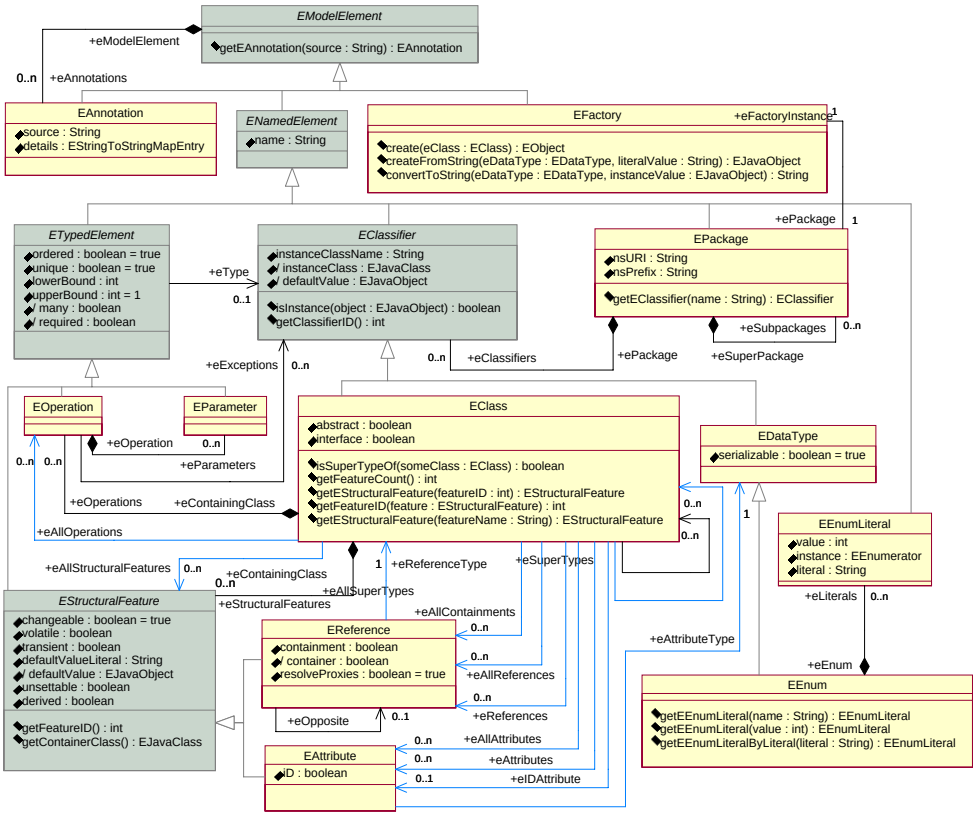

Figure B.1: Ecore Meta-Meta-Model

Appendix C

XMI Representation

C.1 XMI Representation of BPMN Process Model

```
<?xml version="1.0" encoding="ASCII"?>
<bpmn:BProcess xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bpmn="http://denysova.org/bpmn"
    xsi:schemaLocation="http://denysova.org/bpmn bpmn.ecore">
  <elements xsi:type="bpmn:Event" name="Receive Expense Report"
    outputs="//@elements.11"/>
  <elements xsi:type="bpmn:Gateway" name="Account Exists?"
    inputs="//@elements.11" outputs="//@elements.12 //@elements.13"/>
  <elements xsi:type="bpmn:Activity" name="Create Expense Account"
    inputs="//@elements.12" outputs="//@elements.14"/>
  <elements xsi:type="bpmn:Activity" name="Review for Pre-Approval"
    inputs="//@elements.13 //@elements.14" outputs="//@elements.15"/>
  <elements xsi:type="bpmn:Gateway" name="Pre-Approved?"
    inputs="//@elements.15" outputs="//@elements.16 //@elements.17"/>
  ...

  <elements xsi:type="bpmn:Event" name="Inform Employee"
    inputs="//@elements.22 //@elements.23"/>
  <elements xsi:type="bpmn:SequenceFlow" name="1"
    start="//@elements.0" end="//@elements.1"/>
  <elements xsi:type="bpmn:SequenceFlow" name="2 No"
    start="//@elements.1" end="//@elements.2"/>
  <elements xsi:type="bpmn:SequenceFlow" name="3 Yes"
    start="//@elements.1" end="//@elements.3"/>
  <elements xsi:type="bpmn:SequenceFlow" name="4"
    start="//@elements.2" end="//@elements.3"/>
  <elements xsi:type="bpmn:SequenceFlow" name="5"
    start="//@elements.3" end="//@elements.4"/>
  <elements xsi:type="bpmn:SequenceFlow" name="6 Otherwise"
    start="//@elements.4" end="//@elements.5"/>
  <elements xsi:type="bpmn:SequenceFlow" name="7 Amount &lt; 200 "
    start="//@elements.4" end="//@elements.7"/>
  ...

</bpmn:BProcess>
```

C.2 XMI Representation of BPMN Meta Model

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="bpmn"
    nsURI="http://denysova.org/bpmn" nsPrefix="bpmn">
  <eClassifiers xsi:type="ecore:EClass" name="BProcess">
    <eStructuralFeatures xsi:type="ecore:EReference" name="elements" upperBound="-1"
        eType="#//NamedElement" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="NamedElement" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true"
      eSuperTypes="#//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference" name="inputs" upperBound="-1"
        eType="#//SequenceFlow" eOpposite="#//SequenceFlow/end"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="outputs" upperBound="-1"
        eType="#//SequenceFlow" eOpposite="#//SequenceFlow/start"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Activity" eSuperTypes="#//Node"/>
  <eClassifiers xsi:type="ecore:EClass" name="Event" eSuperTypes="#//Node"/>
  <eClassifiers xsi:type="ecore:EClass" name="Gateway" eSuperTypes="#//Node"/>
  <eClassifiers xsi:type="ecore:EClass" name="SequenceFlow"
          eSuperTypes="#//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference" name="start" eType="#//Node"
        eOpposite="#//Node/outputs"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="end" eType="#//Node"
          eOpposite="#//Node/inputs"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="diagram">
    <eStructuralFeatures xsi:type="ecore:EReference" name="bp" upperBound="-1"
          eType="#//BProcess" containment="true"/>
  </eClassifiers>
</ecore:EPackage>
```

Appendix D

Classification of Ecore Meta Model Changes

Element	Operation	A	R	N	Comment
EPackage	Add	✓			
	Remove		✓		content resolvable
				✓	content unresolvable
EClass	Add	✓			
	Remove		✓		Casc. delete on outgoing containment ref.
EDataType	Add	✓			
	Remove			✓	All attributes typed with the concerned data type have to be converted
EEnum	Add	✓			
	Remove			✓	Same as EDataType
EEnumLiteral	Add	✓			
	Remove			✓	All M1 instances with this value have to be converted to another value
EReference	Add	✓			min. cardinality == 0
				✓	min. cardinality > 0
	Remove		✓		All corresp. M1 links have to be removed
EAttrribute	Add		✓		min. cardinality == 0
				✓	min. cardinality > 0
	Remove		✓		All M1 instances have to be removed

Table D.1: Classification of add and remove operations on an Ecore meta-model

Attribute/Reference	A	R	N	Comment
ENamedElement				
name		✓		
EPackage				
nsURI		✓		
nsPrefix		✓		
eClassifiers		✓		
eSubpackages	✓			

Continued on next page

Attribute/Reference	A	R	N	Comment
eSuperPackage		✓		
ETypedElement				
ordered	✓			false⇒true should yield a warning
unique	✓			true⇒false
			✓	false⇒true
lowerBound	✓			decrease
			✓	increase
upperBound	✓			increase
			✓	decrease
eType	✓			new type super class of old type
			✓	any other case
EStructuralFeature				
changeable	✓			
volatile		✓		true⇒false
		✓		false⇒true and lowerCardinality == 0
			✓	false⇒true and lowerCardinality != 0
transient		✓		true⇒false: compute attr. and store value
		✓		false⇒true: delete all values from M1 model
defaultValueLiteral	✓			
unsettable	✓			
			✓	
derived		✓		same as transient
eContainingClass		✓		
ERefence				
containment		✓		M1 instances not contained in other object
			✓	M1 instances contained in other object
resolveProxies	✓			ignored
eOpposite			✓	
EAttribute				
iD		✓		all values of the structural feature are unique
			✓	non unique values exist
EClass				
abstract	✓			true⇒false
		✓		false⇒true: instances of the class not required
			✓	false⇒true: Instances required: not resolvable
interface	✓	✓	✓	same as abstract
eSuperTypes	✓			added/removed super type empty
	✓			added super type contains non-mandatory features
		✓		removed super type contains structural features
			✓	added super type contains mandatory features
eStructuralFeatures	✓			added feature not mandatory
		✓		feature removed
			✓	added feature mandatory
eOperations	✓			
EEnum				
eLiterals	✓			literals added
	✓			literals removed, no refering M1 entities
			✓	otherwise
EEnumLiteral				
value				ignored due to atomicity of enum literals
instance				same as value
literal				same as value
eEnum				same as value

Table D.2: Classification of Structural Featura Changes in Ecore

Appendix E

Change Classes Hierarchy

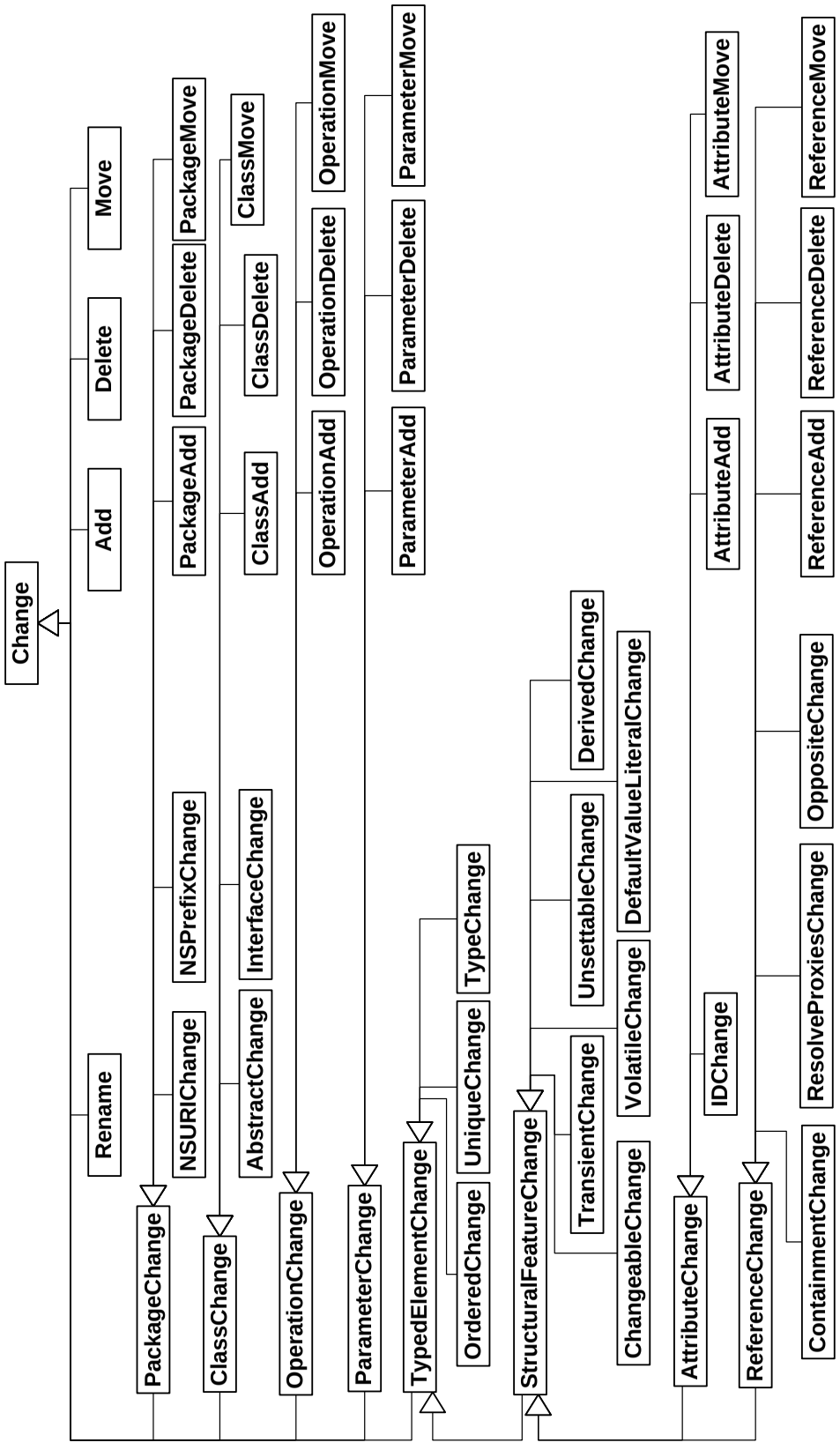

Figure E.1: Change Classes Hierarchy

Bibliography

[AC07] Alfonso Pierantonio Antonio Cicchetti, Davide Di Ruscio. A metamodel independent approach to difference representation. Technical report, Dipartimento di Informatica at the University of L'Aquilal, 2007.

[AHK06] Michael Altenhofen, Thomas Hettel, and Stefan Kusterer. OCL support in an industrial environment. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 2006.

[AMW] Eclipse.org, Atlas Model Weaver (AMW). http://www.eclipse.org/gmt/amw/.

[AP03] Marcus Alanen and Ivan Porres. Difference and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[ATL] Eclipse.org, Atlas Transformation Language (ATL). http://www.eclipse.org/m2m/atl/.

[Bar93] Ludwig Von Bartalanffy. *General System Theory: Foundations, Development, Applications*. Georges Braziller, Inc., 1993.

[BD96] Michael Bertol and Volker Diekert. Trace rewriting: Computing normal forms in time o(n log n). In *STACS '96: Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, pages 269–280, London, UK, 1996. Springer-Verlag.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000.

[BG07] Richard F. Paige Boris Gruschko, Dimitrios S. Kolovos. Ctowards synchronizing models with evolving metamodels. In *Workshop on Model-Driven Software Evolution (MODSE 2007)*, Amsterdam, the Netherlands, 2007. 11th European Conference on Software Maintenance and Reengineeringn.

[BGGK07] Steffen Becker, Thomas Goldschmidt, Boris Gruschko, and Heiko Koziolek. A process model and classification scheme for semi-automatic meta-model evolution. In *Workshop MDD, SOA und IT-Management (MSI2007)*, 2007.

[BHJ+00] Philip A. Bernstein, Laura M. Haas, Matthias Jarke, Erhard Rahm, and Gio Wiederhold. Panel: Is generic metadata management feasible? In *The VLDB Journal*, pages 660–662, 2000.

[BPD] Business Process Definition Metamodel (BPDM). http://www.omg.org/.

[BPM06] Business Process Modeling Notation (BPMN) Specification. http://www.bpmn.org, May 5 2006.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison-Wesley, Boston et. al., 2000. ISBN: 0-201-30977-7.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[EMF] Eclipse.org, Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

[FB03] Ed Merks Raymond Ellersick Timothy J. Grose Frank Budinsky, David Steinberg. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Professional, 2003.

[Fen94] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.

[Gir06] Martin Girschick. Difference detection and visualization in uml class diagrams. Technical report, TU Darmstadt, 2006.

[GME] The Generic Modeling Environment (GME). http://www.isis.vanderbilt.edu/projects/gme/.

[Goo04] Paul Goodman. *Software Metrics: Best Practices for Successful IT Management*. Rothstein Associates Inc, 2004.

[Gru07] Boris Gruschko. Changes classification in M2 models. In Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven, editors, *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 277–280. GI, 2007.

[Här69] Holden Härtl. *Implizite Informationen:Sprachliche Ökonomie und interpretative Komplexität bei Verben*. PhD thesis, Humboldt-Universität zu Berlin, 07 1969.

[Jac86] Ivar Jacobson. Language support for changeable large real time systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 377–384, New York, NY, USA, 1986. ACM Press.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Jou05] Frédéric Jouault. Loosely coupled traceability for atl, 2005. ISBN=82-14-03813-8.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lay06] Henry Laycock. Object. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2006.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[LB06] Linda M. Laird and M. Carol Brennan. *Software Measurement and Estimation: A Practical Approach (Quantitative Software Engineering Series)*. Wiley-IEEE Computer Society Pr, 2006.

[Leh69] M. M. Lehman. The programming process. Research Report RC 2722, IBM T. J. Watson Research Center, Yorktown Heights , NY , USA, September 1969.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *EWSPT*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[Lie06] Benjamin A. Lieberman. *The Art of Software Modeling*. Auerbach Publications, Boston, MA, USA, 2006.

[mat] Eclipse.org, AMW Use Case - Matching. http://www.eclipse.org/gmt/amw/usecases/matching/.

[McM95] Paul E. McMahon. Pattern-based architecture: Bridging software reuse and cost management. *CrossTalk - The Journal of Defense Software Engineering*, 1995.

[MEd] MetaEdit+, Domain specific modeling tool. http://www.metacase.com/.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*, volume 2967 of *Lecture Notes in Computer Science*. Springer, 2004.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[MOF06] Meta Object Facility Core Specification version 2.0. http://www.omg.org/, 2006. formal/06-01-01.

[NZR07] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. Towards traceability of model-based testing artifacts. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 105–114, New York, NY, USA, 2007. ACM Press.

[OMG] Object Management Group (OMG). http://www.omg.org/.

[Pre00] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2000.

[Rac95] L. B. S. Raccoon. The chaos model and the chaos life cycle. *Software Engineering Notes*, 20(1):55–66, January 1995.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[RL06] Romain Robbes and Michele Lanza. Change-based software evolution. In *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pages 159–164, 2006.

[Roy70] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *WESCON Technical Papers, v. 14*, pages A/1–1–A/1–9, Los Angeles, August 1970. WESCON. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, 1987, pp. 328–338.

[SK04] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *J. Vis. Lang. Comput*, 15(3-4):291–307, 2004.

[SLKN01] Jonathan Sprinkle, Ákos Lédeczi, Gabor Karsai, and Greg Nordstrom. The new metamodeling generation. In *ECBS*, page 275. IEEE Computer Society, 2001.

[Spr03] Jonathan Mark Sprinkle. *Metamodel driven model migration*. PhD thesis, 2003. Director-Gabor Karsai.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[Swo00] Chris Swoyer. Properties. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2000.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–304, New York, NY, USA, 2007. ACM Press.

[Whi06] Stephen A. White. Introduction to BPMN. Whitepaper, October 16 2006. http://www.bpmn.org.