CHALLENGES IN THE INTEGRATION OF DOMAIN-SPECIFIC AND SCIENTIFIC BODIES OF KNOWLEDGE IN MODEL-DRIVEN ENGINEERING

ERIC MIOTTO

Supervisor
PROF. TULLIO VARDANEGA

Tesi per la laurea specialistica in Informatica

Dipartimento di Matematica Pura ed Applicata
Facoltà di Scienze Matematiche, Fisiche e Naturali
Università degli Studi di Padova

September 2009

CHALLENGES IN THE INTEGRATION OF DOMAIN-SPECIFIC AND SCIENTIFIC BODIES OF KNOWLEDGE IN MODEL-DRIVEN ENGINEERING

September 2009

CANDIDATE Eric Miotto _____

SUPERVISOR prof. Tullio Vardanega _____

ABSTRACT

Through the use of models and transformations and the recognition of the importance of application domains, Model Driven Engineering (MDE) promises to raise the quality of software systems and to lower the development effort. However, the successful leverage of Model Driven Engineering is fraught with several important problems and choices: we must be aware of these and tackle them to obtain a development environment that is effective, sound and affordable.

With this concern in mind we address two important problems:

- the definition of Domain Specific Languages, which express domain concepts and form the foundation for the implementation of all the features of the development environment (editors, transformations, serialization, view management, ...);
- the complexity of proving the correctness of transformations. This complexity is high and we need to tame it to the maximum extent possible.

We first discuss these problems from a scientific point of view and we propose some reasoned solutions. Next we explore the feasibility of these solution by evaluating whether and at what cost current available technologies in Eclipse permit to implement them.

PUBLICATIONS

Some ideas and figures (in particular the content of chapter 3) have appeared previously in [1].

The technical investigation discussed in chapter 4 has been inserted in [2] and submitted to DATE2010.

CONTENTS

1 INTRODUCTION

In September 2004 a research project called ASSERT (Automated proof-based System and Software Engineering for Real-Time applications) [3] was started, partially funded by the European Commission under the Sixth Framework Programme (FP6). ASSERT lasted for three years, was led by the European Space Agency and was participated by industrial and academic partners. The aim of this project was the definition of a development process for high-integrity and real-time systems such that it could guarantee the respect of non functional requirements (for example temporal deadlines or memory occupation) not through ad-hoc proofs done after the construction but through a construction approach that is proved correct once and for all.

Indeed, in current approaches to software engineering, construction techniques deal only to the mere implementation but not with the verification of the satisfaction of requirements, and the fulfillment of requirements can be proved only once the system is implemented. For these reasons, the employed verification techniques are usually general and thus not very informative (e.g. testing can find bugs but cannot prove their absence). Even if some ad-hoc techniques specific for system are devised (with the shame that they cannot be reused with other systems), the point is that the system is already implemented and it is needed to return again to its design and construction to fix the problems – this can be very expensive and there is the risk to introduce new unanticipated bugs. We can name this approach as *construction-by-correction* [4] – a working system is obtained by means of repeated constructions and verifications.

On the contrary, in other mature engineering fields, the system (let it be a bridge or a chip) is not immediately built, but it is first modeled using patterns and solutions apt for its class, devised so to permit the verification of relevant properties. This way it is not necessary to build the system to detect its defects, but they can be found in the design phase, with the great advantage that it is known exactly how to deal with them and that further problems will be properly detected thanks to the employed patterns and solutions. The construction of a system is started only when the design meets all the requirements (to a certain degree). We can refer to this approach as *correctness-by-construction* [4] – a working system is built by means of proven modeling and design. So, to coin a slogan, ASSERT wanted to devise a correctness-by-construction approach to replace the construction-by-correction approach largely employed in software engineering.

Among its results, ASSERT recognized the suitability of Model Driven Engineering (MDE) for the construction of real-time software [5]. Model Driven Engineering advocates the construction of better software with less effort through: (i) the use of models for software specification in place of source code; and (ii) the use of automated transformations to obtain the final system from these models. (The terms "model" and "transformation" will be defined more precisely in chapter 2, for now their intuitive meaning should suffice.)

ASSERT also showed that MDE should be applied in a proper way in order to achieve its maximum effectiveness. In fact, the mere use of models and transformations is not sufficient to harvest the benefits of MDE. For example, at first glance we may think that a system should be described by a single model. This approach is apt for small systems, but for large ones the model would cram too many concerns and this renders its specification and its comprehension challenging and difficult. If this and other problems are not tackled, MDE can be more expensive than traditional approaches.

In March 2009 the CHESS project (Composition with Guarantees for High-integrity Embedded Software Components ASsembly) took up from the conclusions of ASSERT in order to realize a fully functional prototype of an MDE environment for high-integrity real-time systems. It should be quite clear that in CHESS it is needed to employ methodologies that are not only sound from a scientific point of view; it is required to implement them using current available technology and without spending too much development effort. Indeed a costly approach can hamper the effectiveness, the maintainability and the longevity of the environment.

In the context of CHESS, we are concerned with the construction an MDE development environment with the following two characteristics:

1. *sound leverage of application domains*. Software systems should be specified using concepts of the application domain of interest and not through generic computation concepts. Nevertheless, we should be aware that different application domains can share common concerns and we should strive to share also their solutions;

2. *guarantee of correctness-by-construction in a sound, provable and affordable way*. We should ensure statically that generated applications respect their non functional requirements.

In particular, we recognize two important problems:

- *the definition of Domain Specific Languages (DSL) for the specification of models*. These languages are crucial because: (i) they permit to express concepts of the particular application domain (whence their name); and (ii) they influence the construction of the rest of the development environment (editors, transformations, serialization, ...). They should be defined properly and in a way that guarantees longevity. At the present time we can choose to define DSL using UML 2.x profiles or metamodeling;

- *the complexity of proving the correctness of transformations*. Especially in the context of real-time systems, we need to ensure that transformations do not introduce arbitrary semantics in output models. We say that a transformation from a model A to a model B is correct if: (i) the semantics of B contains the semantics of A; (ii) the semantics we add to B does not contradict the semantics contained in A.

  Proving the correctness of transformations is intrinsically a daunting task. Anyway we want to construct our environment in order to avoid accidental increments in the complexity of these proofs,

and so we want to find the factors that can have the greatest impact on complexity and treat them appropriately.

These problems must first be discussed from a methodological point of view, in order to find out solutions that have a scientific support. Next it is necessary to verify whether, to which degree and with which effort the current available technology enables us to implement these solutions. A sound solution that requires too much effort to be implemented and maintained is not attractive and it may hamper the effectiveness of the development environment.

1.1 THESIS ORGANIZATION

The thesis elaborates on these problems, their proposed solutions and the support that we found in state-of-the-art tools. We start in chapter 2 with an extended introduction to Model Driven Engineering. Firstly we explain the tenets of MDE, in particular the importance of application domains, models and transformations; secondly we describe Model Driven Architecture (MDA), a particular MDE initiative that we refer to in the next chapter; thirdly we discuss about the nature of application domains, underlining the need to recognize transversal concerns and the necessity to resolve them once and for all using scientific methods.

In chapter 3 we elaborate on the two problems we stated above, arguing their appropriateness and devising if possible some ways to address them. The tone of this chapter is scientific and methodological.

To determine the concrete feasibility of our proposals, in chapter 4 we study technologies available in Eclipse and we evaluate whether and to which extent they permit to implement the solutions devised in the previous chapter.

Finally, in chapter 5 we sum up our work and highlight the main contributions.

1.2 NOTATION

In the thesis we adopt the following typographical conventions:

- URL are in brown typewriter fonts, e.g. `http://www.uml.org/`;
- section, list, page, glossary and figure references are written in blue font, e.g. chapter 1;
- bibliography references are written in green font, enclosed in square brackets, e.g. [3];
- references to part of cited documents are written with the Kurier font [6], e.g. metamodelReference;
- for code we use sans serif text with syntax highlighting, e.g. `int i=5;`;
- file names, paths, menu items and buttons are written in typewriter font, e.g. `/usr/local/` .

2 MODEL DRIVEN ENGINEERING

This chapter begins with a detailed description of Model Driven Engineering; next we present Model Driven Architecture, a particular MDE approach from which we borrow concepts and terminology; in conclusion we discuss about the role of scientific bodies of knowledge in complementing application domains.

2.1 AN OVERVIEW OF MODEL DRIVEN ENGINEERING

In the development of software we are continuously challenged by the demand for increased productivity, high quality and shorter time-to-market. In other words, as time passes we must develop software that: (i) contains a larger number of functionalities; (ii) fulfills its specification to the maximum possibile extent; and (iii) requires less effort and time to be produced and maintained.

At present, software is implemented using programming with more manual work than automation. Current programming languages are quite effective to facilitate manual programming, since they abstract from low level computation details (such as the architecture of processors) and provide powerful concepts and utilities (like classes and libraries) that can be used to render concepts of an application domain. Nevertheless, this approach has two main shortcomings:

1. despite the high level of abstraction, we are still bound to the computation domain and this makes it quite difficult to translate problem concepts into proper concepts of programming languages. We can obviously express domain specific concepts with them, but the semantics of the application domain – the one we are ultimately interested in – is buried under the semantics of computation. Hence program comprehension and verification become more problematic and – even worse – it's difficult to relate software components to the requirements they are meant to satisfy;

2. we are bound to the employed technology and thus changing it requires great manual effort on updating the code, even though the scope of the system remains the same. In fact, programming languages permit to change the underlying processor architecture without major problems, but this is not enough.

Model Driven Engineering (MDE) [4, 7, 8] is an approach to software engineering that addresses these two problems with the additional goal to guarantee increased productivity and increased quality. In particular MDE promotes:

1. the use of models at various levels of abstraction described with Domain Specific Languages (DSL) as a vehicle for system specification;

2. the use of automated transformations to progressively turn the user model into a software product ready for final compilation, binding and deployment.

In the next two sections we explain in detail how these two characteristics allow to tackle the highlighted deficiencies of current approaches.

2.1.1 *Models as means to specify systems*

MDE proposes models as first-class artifacts for software development. A model is a simplification of a system that can be used in place of this to verify some properties of interest. The use of models is common in engineering disciplines, since they permit to represent systems in a simplified way, which in turn enables the user to reason more easily about properties of the final products. Moreover, they allow the user to validate the design especially in the early phases, thus avoiding to produce and waste prototypes only to detect small design flaws. According to this definition, we might argue that: (i) source code can be considered a model (since it represents at high level an executable system written in machine code); and (ii) models are already employed by practitioners, thanks to the popularity of Unified Modeling Language (UML). In fact, in MDE the word model is used in a narrower and more specific meaning:

(i) source code is surely a model but it lacks an important characteristic of MDE models: understandability [7]. In other words, models should express in a clear way the main concerns of the system under study: users should put little parsing effort to understand it. As we mentioned before, source code doesn't show clearly the functions of a software system, as computation concepts and abstractions tend to prevail. This fact hampers the comprehension of the system, especially by the stakeholders who, as domain experts, could notice incongruences more readily if we could provide them with a model that expresses distinctly the concepts of the application domain. Thus, in MDE models should mainly contain domain concepts and for this reason source code is not considered a model;

(ii) nowadays models are often employed in software development but they tend to be used as support to the documentation or as mere guideline to the programmer on how to produce the final code. Moreover, syntax and semantics of these models are often stated in a rather informal way. In MDE instead models have a central role, as they guide the entire development process – and thus their syntax and semantics should be defined precisely and without ambiguity.

To summarize, models in MDE should state requirements and functionalities of software with explicit reference to the application domains of interest, so that they emerge clearly without being obfuscated by concepts unnecessary for their comprehension, like computation concepts. This way models can be even manipulated directly by domain experts without any intermediation of software developers: software specification will consequently be more aligned with users' needs.

As a consequence, models in MDE have a higher longevity than source code. Indeed, source code tend to rapidly become obsolete, and most of the times this is not due to changed requirements but to technological reasons, independent from the functionalities. Common causes of source code obsolescence are changes in the interface

of third party libraries and the adoption of a new software platform. Instead a model contains only domain concepts, and thus its content will remain unchanged and valid despite all the technological changes we might experience during software development. Let us note that we must alter the model to accomodate new or revised requirements; but this is necessary and desired, while we want to avoid changes to the model due only to technological reasons.

Models are specified using languages – much like source code. Since modeling languages contains concepts of application domains, they are called Domain Specific Languages (DSL). Their definition is extremely important and should be done very thoroughly, because the understandability and the technological independence of a model are determined by how well the DSL let us describe the system with respect to the application domain. For this reason in the remainder of this section we discuss this topic in depth.

In which way should we specify a modeling language? Without going into unnecessary details, a language is simply a set of valid "sentences" – a "sentence" can be a program for the C language or a class diagram for the UML language. The problem is that the number of these sentences is infinite and we can't enumerate them, thus we need to describe the rules to construct valid sentences. A model serves well this purpose since we need to abstract from the fact that the language is an infinite set: for example, grammars and finite state automata are commonly used to specify the syntax of a language and can be regarded as models.The model used to describe a modeling language is called a metamodel (figure 2.1) and it is made up of:

- the *abstract syntax*, which specifies the concepts we can talk about and their relationships. The abstract syntax might include also additional explicit constraints;
- the *concrete syntaxes*, which specify in which way the abstract syntax is represented and stored. A concrete syntax can be textual or graphical;
- the *semantics*, which specifies the meaning of each concept. Perhaps this is the most important aspect of the metamodel.

It is now opportune to provide an example to clarify the concepts related to metamodels. A political map of the world can be considered a model of political geographical assets of the planet. A limited but reasonable metamodel for the map can be defined as follows (figure 2.3):

- its abstract syntax is made up of the concept of state, of the relationship of borders between states and of the constraint that two states cannot overlap;
- its concrete syntax is the usual graphical syntax of maps, in which states are colored regions and borders are black lines between states;
- its semantics asserts that a state represents a physical region of the planet which, in proportion, has the same shape and position and that a border represents a physical contact between two physical regions.

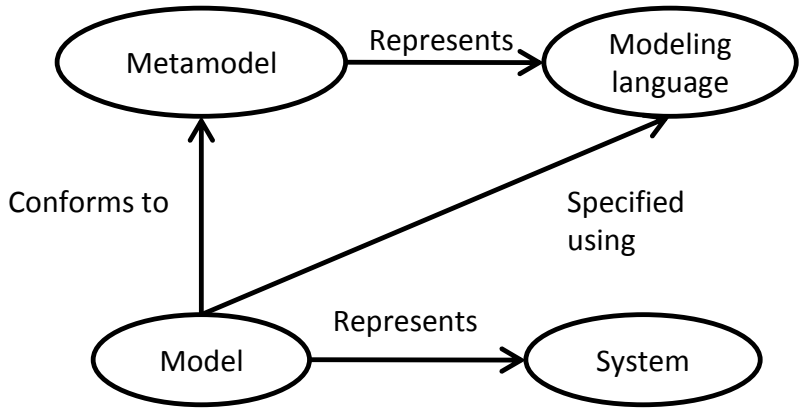

Figure 2.1: A *model* represents or specifies a system in which we are interested. This model should be specified using a language, that contains the concepts and relationships of the domain of interest. Quite logically in the context of MDE, the language is specified using a model, which is called *metamodel* to underline that it talks about the way models are specified. For the sake of simplicity, we can say that a model is conformant to a given metamodel, without citing explicitly the language that "stands" between the two. This figure is inspired by [9].

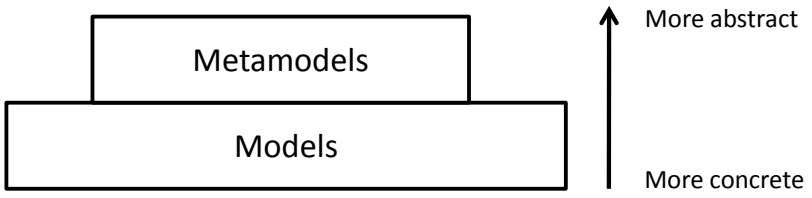

Figure 2.2: The relationships and characteristics of models and metamodels can be effectively depicted with a pyramid. At the bottom we find the models, which are concrete in that they represent systems we are directly interested in (e.g. a software); at the top we find the metamodels, which are somewhat abstract since they are used to establish the concepts and the relationships we can use in models. The level of models has a wider width that the one of metamodels because we need and produce more models than metamodels. This figure is inspired by [9].

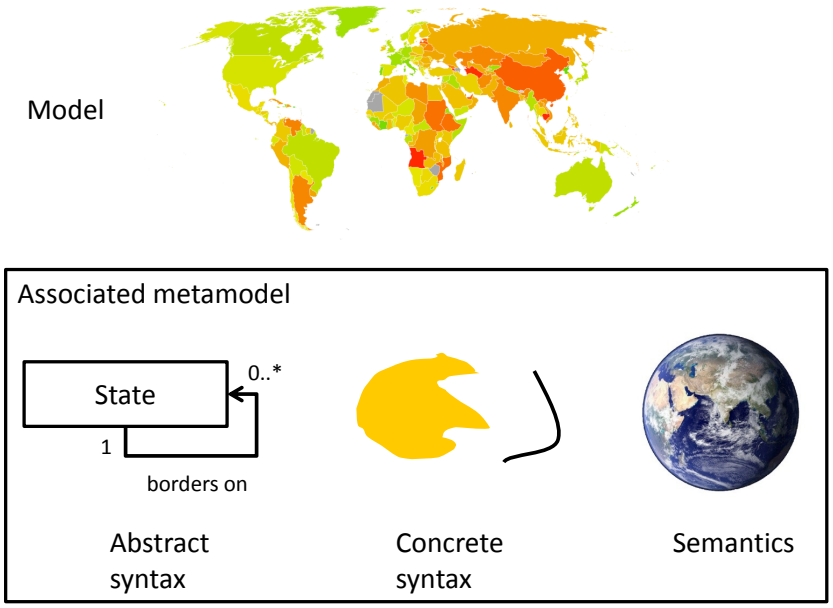

Figure 2.3: A concrete example of a metamodel and its associated elements for a political map of the planet. The abstract syntax, represented with an UML class diagram, is made of the the concept of state and the relationship of bordering; the concrete syntax is composed of colored regions and lines; the semantics maps the concepts to the physical entities of the Earth. (Political map obtained from http://gunn.co.nz/map/; Earth image courtesy of NASA Visibile Earth, http://visibleearth.nasa.gov/view_rec.php?id=2429)

The name metamodel intuitively suggests that this model has a specific purpose in that talks about models – that is it is used to specify how models of a certain kind are built, while usually a model is used to describe concrete systems like a software, a house or a car. For this reason, metamodels are considered more abstract than regular models. It's also reasonable that the number of metamodels we may need is less that the number of models we specify: indeed with a single language we express multiple models. Thus we can assume to have a little pyramid (figure 2.2), with the models at the bottom and the metamodels at the top.

Since metamodels are themselves models, we need a language to specify them. We can continue this reasoning endlessly, and we can expect that the more we raise the level of abstraction the less models we need (figure 2.4). From these statements we understand that in the context of MDE it does not make sense to add too many levels, instead we should stop and provide at the highest level a model that is able to express, in addition to models at the bottom level, all the models that pertain to its level including itself. We could do this inside the metamodels – UML will be a suitable language (in fact we just need its class diagram) but this will be exceedingly confusing. MDE researchers prefer to add a level above metamodels in which we put only a specialized model that is used to describe the metamodels and itself. Sometimes this model is called metametamodel. This might sound weird at first,

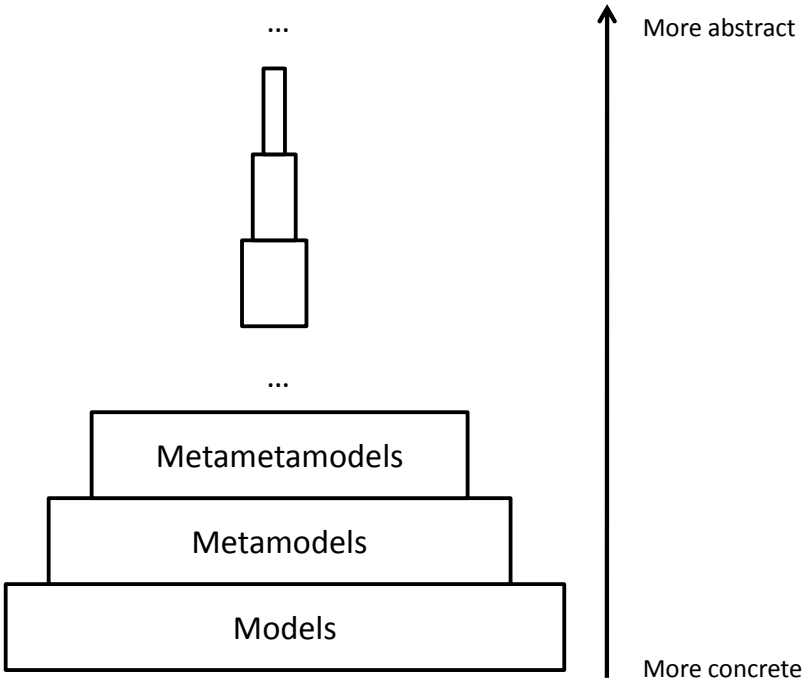

Figure 2.4: The pyramid containing models and metamodels can become infinitely tall. As we raise the level of abstraction, the number of models in each individual will decrease and they become increasingly less useful.

but we should recall that for example English dictionaries and grammars, models of the English language, are themselves written in English. All this discussion is summarized in figure 2.5.

To recapitulate, we present a concrete example to better illustrate the concepts of model, metamodel and metametamodel, which is also depicted in figure 2.6. Let us consider a (tiny) software system: this can be represented using a component diagram, which thus has the role of the model. This diagram contains specific components and interfaces, for example the component foobar. The component diagram is specified with the UML language: the latter is specified through the metamodel described in the UML superstructure [17]. For instance, in this metamodel we find all the allowed entities in a component diagram, like the component and the interface. The concrete component foobar can be seen as a concrete instance of the component entity of the UML metamodel. The UML metamodel is described using the Meta Object Facility (MOF) language, an OMG specification for the specification and management of metamodels. In turn, MOF is described with a metametamodel, which contains the allowed concepts we can express in the UML metamodel. MOF has the concept of class that is used to render the component and interface entities in the UML metamodel.

2.1.2 *Transformations as means to obtain implementation*

In spite of their utmost importance, models alone are not sufficient: we need a way to automatically generate the final system from the various models we may specify. We can do this manually, similarly to

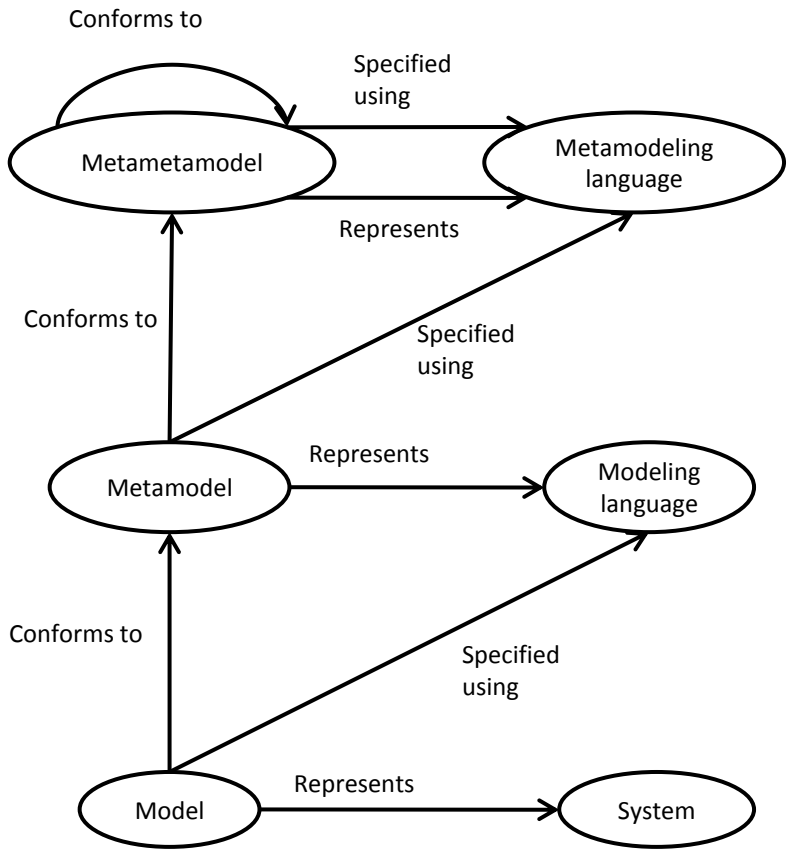

Figure 2.5: Generalization of figure 2.1 that acknowledges the necessity for a language to specify metamodels and thus also for a metametamodel to define this language. As highlighted by figure 2.4, we need only this metametamodel and there is no need to continue further: thus the metametamodel is used to describe itself, as English dictionaries and grammars are themselves written in English. This figure is inspired by [9].

what happens when we use models informally; but this way we lose control on how the features of a model are implemented, with the risk of obtaining suboptimal solutions and – even worse – with the risk that similar parts of the model are translated with different code.

MDE instead advocates the use of transformations, which given one or more models and some parameters generate one or more models at a lower level of abstraction, used for specifying additional implementation detail (execution platform, ...) or for specific purposes (for example static analysis). In addition a transformation could output source code that implements the system.

Thus transformations help ensure that produced artifacts are consistent with the input models. Moreover, they contribute to augmenting the longevity of the models with regard to technology: in fact if we need to change the underlying platform or we need to target multiple platforms, we only need to change the transformation or devise new

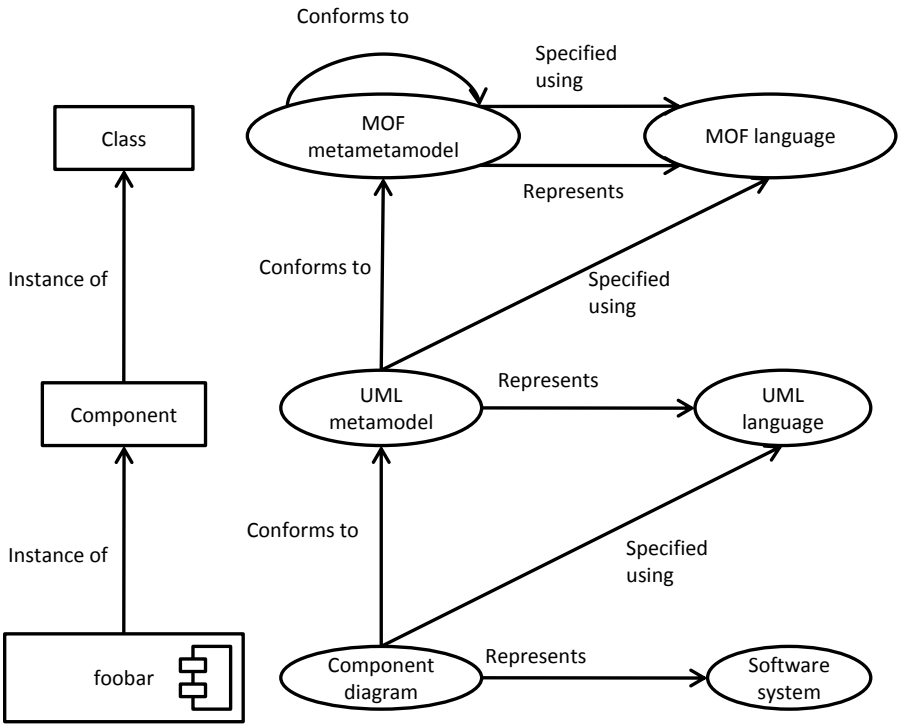

Figure 2.6: Instantiation of figure 2.5 to a concrete example. A software system can be described with a component diagram, which is its model. This diagram contains concrete components and is specified using UML, which is described using the metamodel we find in OMG specifications. That metamodel defines the concept of component. In turn, the UML metamodel is specified using the MOF language, which is described using the MOF metametamodel. For instance, this metametamodel contains the concept of class, used to model the concept of component in the UML metamodel.

ones, without touching the model, which is inherently durable because it express only domain concepts.

2.2 MODEL DRIVEN ARCHITECTURE

MDE is stated in a quite general way, without dictating any particular approach – we are not told which models to use to specify a system and which transformations to employ to obtain the final implementation. Instead it is useful to refer to a particular approach in order to better discuss about difficulties in the adoption of MDE and about their solutions – for example, it's likely that we have different models used with different roles (e.g. specification or analysis) and thus addressing different problems. If we don't assume a specific implementation approach, it's difficult to see the obstacles that stand before us; on the other hand, we incur a loss of generality in our discussion since we don't consider MDE as a whole and we might ignore some aspects, but the conclusions we draw from the analysis of the approach can be surely applied to MDE.

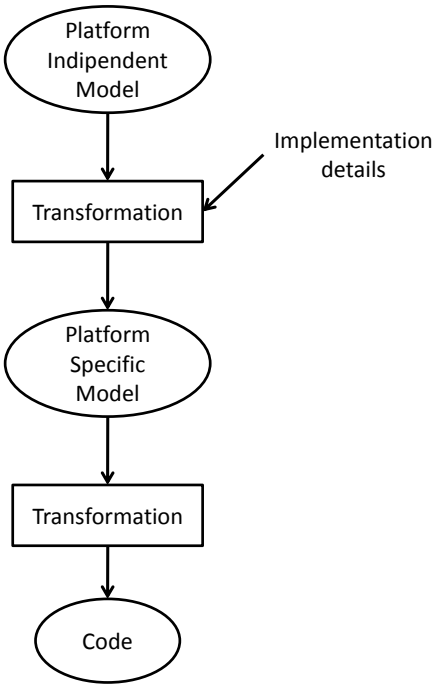

Figure 2.7: In MDA the system is specified using a Platform Indipendent Model (PIM), which abstracts away from implementation details. In order to obtain a system that can be turned into a deployable executable, we have to transform the PIM into a Platform Specific Model (PSM). This transformation requires guidance information on what implementation choices to make. A final stage of transformation is required to turn the PSM into source code.

The particular MDE approach we have chosen is the Model Driven Architecture (MDA) standard [10, 11], born in 2001 and endorsed by the Object Management Group (OMG). MDA also inspired the research inside ASSERT and CHESS. OMG noticed that software developers are struggling to implement over and over again their products against new emerging software platforms (like CORBA, Java and .NET). While it's unlikely that only one software platform emerges as standard, they spend superfluous effort in coding the functionalities of software not because they are changed but because they need to be implemented differently per platform. Thus MDA proposes the use of models and trasformations to effectively separate implementation details from functionalities in the specification and production of software, in order to allow easy change of technology (e.g. from .NET to Java) while salvaging and reusing the durable information about what software should do.

To this end, MDA advocates the following approach (figure 2.7):

1. we first construct a Platform Independent Model (PIM) that specifies the solution in a way that does not depend on any particular implementation (or platform in MDA terms). The PIM expresses only the functionalities of the software and thus its validity will last longer than traditional source code;

2. we then transform the PIM into a Platform Specific Model (PSM), by feeding the transformation with information about the chosen execution platform and its characterizing parameters. The PSM embodies implementation details but it's not the final system. We assume that the PSM is automatically generated and users are not allowed to directly modify it;
3. we perform a range of analyses on the PSM to validate its feasibility and adequacy against a range of criteria. Some analyses are meaningful only when we take into account the specific characteristics of the chosen platform: for example schedulability analysis requires to know the run-time overhead parameters of the real-time kernel (time required for a context switch, time to execute an interrupt, ...). The PSM surely contains this information and is expressed in a way that is more amenable to analysis than source code.;
4. when satisfied with the results of the analyses, we launch the final stage of transformation from PSM to code to obtain the final system, otherwise we return to the PIM.

For the sake of completeness, in MDA the PIM should be derived (manually or automatically) from a model called Computation Independent Model (CIM), which represents the application domain of interest. We note that in [10] the discussion is focused on PIM and PSM, while CIM has little coverage – this is coherent with the interest of MDA in addressing technological variability.

In addition to methodology, MDA dictates the technologies to employ: in particular the languages used for both PIM and PSM should be specified with Meta Object Facility (MOF), with the possibility to use UML and the profile mechanism. When we refer to MDA, we will not require the use of these technologies; nevertheless, MOF and UML have the huge advantage of being standard and vendor-neutral and thus they tend to be the first options to be considered in the realization of an MDE approach.

Some people criticize MDA because it dictates a strict approach and uses models mainly to address technological variability, with a very little accent on application domains (for example [12] and [13]). In our opinion, MDA is a good approach for our discussion; moreover some aspects as the recognition of application domains can be easily fitted in it, as we will show in the next section.

2.3 ACKNOWLEDGING SCIENTIFIC BODIES OF KNOWLEDGE

Before we continue with the main content of this thesis, we would like to make an important remark about the nature of application domains.

MDE recognizes that in order to raise substantially the level of abstraction of development the model should not talk about computation but about the problem, about the application domain. That is, we should not use programming languages or general purpose languages, but instead we should adopt DSL that model directly the domain of interest.

At first glance it might seem that each application domain does not overlap with the others, so that each domain has concepts and solutions completely specific to it, without significant opportunity of reuse. To counter this wrong impression, let us consider for example the construction of software for automotive, railway and aerospace domains.

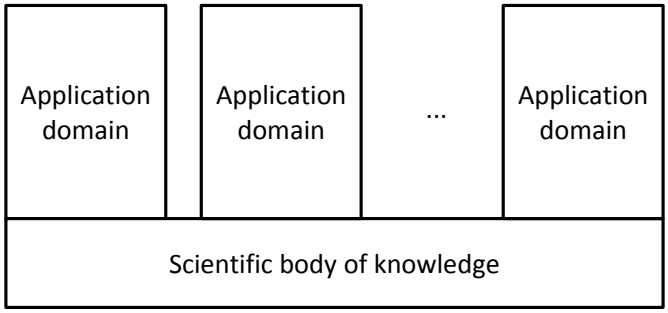

Figure 2.8: A scientific body of knowledge contains notions and concepts that are transversally employed by several application domains.

We are clearly considering three different application domains with different concerns – the functions needed in a car are clearly different for those needed in a train or in a plane. Yet, these application domains face common concerns, (for instance) in that parts of their software must demonstrably meet temporal deadlines.

Hence application domains do share common transversal problems with other domains. Nevertheless, we might think that these problems can be resolved by each domain on its own – surely we have duplication of effort and solutions but the importance of application domains in MDE might seem to justify this conduct. To an extreme degree, some argue that different application domains in fact may need different solutions [14]. We believe that the nature of such transversal problems is such that their proper recognition and treatment require knowledge that is more scientific than domain specific. This might sound irrelevant, but a scientific approach gives higher guarantees that the solution will be sound, solid and cost effective.

Let us return to the example of automotive, railway and aerospace domains. When we consider timeliness and predictability problems, we ought to know that the real-time scientific community has long known those problems and devised a range of analytic and engineering techniques to best cope with them. Such techniques may be regarded as best practices that form a so called scientific body of knowledge. For example, in the real-time literature we can find many schedulability analyses conceived for fixed priority systems, each of them adding greater sophistication to the previous one. If we look at these techniques, we can see that the knowledge required to construct them is vast (actually there is also mathematics involved) and that extensive community discussion is needed to assess their aptness. A similar result cannot be obtained inside an application domain, because practitioners and researchers have different job descriptions, different background and different objectives.

Thus we firmly believe that the domain specific knowledge should be complemented with adequate doses of scientific body of knowledge (figure 2.8), otherwise we may adopt suboptimal solutions that diminish the value of the system.

To summarize, both application domain knowledge and scientific bodies of knowledge are crucial to effective MDE and they should both guide and steer the application development. In the MDA approach in

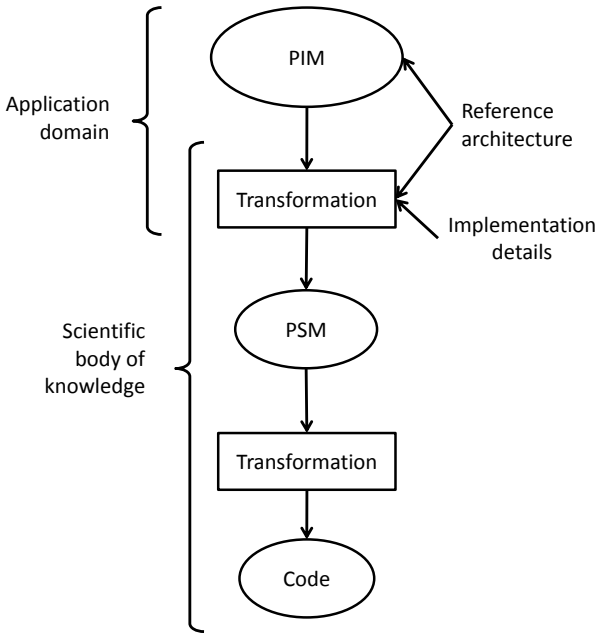

Figure 2.9: The way application domain knowledge and scientific knowledge fit in MDA. PIM and its languages are influenced by the application domain; PSM and its languages are influenced by the scientific body of knowledge; the application domain may impose a reference architecture; the transformation from PIM to PSM is determined by both application domain and scientific bodies of knowledge; analyses of and code generation from the PSM are influenced by the scientific body of knowledge.

particular, those two complements of knowledge have the role and impact illustrated in figure 2.9:

- they determine how the DSL to use for PIM and PSM are defined. At PIM level we must be able to conveniently express concepts that pertain to the application domain. Conversely, at PSM level we must be capable of constructing the system using methods that have a scientific pedigree and background;
- the application domain often requires to deploy applications in accord with some estabilished reference architecture, which may be physical as well as logical;
- the transformation of the PIM to PSM is guided by both application domain and scientific knowledge. The former retains design choices favored by the user (or the domain culture), while the latter permits to implement them in a scientifically based manner;
- the allowable range of analyses to be performed on the PSM and the practices of code generation from it must be rooted in the scientific body of knowledge.

2.4 SUMMARY

In this chapter we have introduced the fundamental notions that we need to discuss in the next chapter the problems we encounter in incorporating MDE in the development process.

First we presented Model Driven Engineering itself as an attractive approach to software engineering, which promises to be more effective than programming languages and coding in pursuing productivity and quality in the face of increasing richness of functionalities of software systems. In this sense MDE advocates:

1) in place of source code and informal models, the use of models specifies with Domain Specific Languages (DSL), which describe software systems in terms of application domains instead of in terms of computation concepts;

2) in place of manual coding and manual translation of requirements, the use of automated transformations that progressively generate the final implementation from the models.

MDE does not dictate in which way models and transformations should be employed. Nevertheless in order for us to be able to see the problems in applying MDE we need to single out and discuss a specific, concrete approach. To this end, we discussed Model Driven Architecture, an MDE approach created and endorsed by OMG, which has a strong accent on achieving independence from software platforms, so that change of implementation can be done without ditching all the implementation artifacts. MDA proposes to specify a system with a Platform Independent Model (PIM) that abstracts from the particular platform and contains only the specification of functionalities. Once we choose the executing platform and its parameters, we can transform the PIM into a Platform Specific Model (PSM), which contains implementation details and permits the use of analyses to assess the suitability of the design according to some criteria. If we are satisfied with the resulting PSM, with another transformation we can obtain the code, otherwise we return to the PIM to change it accordingly.

At the end of the chapter, we expanded on application domains. While it's acknowledged that each application domain has concerns that require specific recognition and treatment, we argue that some concerns are shared among domains and we can't tolerate they are tackled separately by each interested domain. Indeed the recognition and treatment of these shared concerns require the support of scientific method to find sound and cost-effective solutions: we need deep knowledge to confront with them and we need extensive discussion to assess the suitability of proposed solutions. So we contend that domain specific knowledge should be complemented by scientific knowledge in order to achieve the maximum benefit in MDE.

3 APPLYING MDE IN THE REAL WORLD

The mere adoption of MDE alone is not sufficient to improve the quality of development of software systems. The basis of MDE may seem detailed enough to realize an effective development environment – we need only to define the languages and the transformations and we are done. In fact the definition of languages and transformations are fraught with difficulties that, if ignored or treated lightly, can diminish the benefits brought by MDE or can raise the effort and the cost of realization and maintenance of the development environment. In particular two problems stand before us in the attainment of this goal; in the remainder of this chapter we want to elaborate on their essence, background and trade offs:

1. DSL seem to be important vehicles for the expression of both the application domain and the scientific bodies of knowledge. Given the coverage that both elements have in the deployment of MDA, DSL should serve for both PIM and PSM. The DSL should be carefully designed, for they determine the effectiveness at the user level, the affordability of the implementation and the durability of the infrastructure. DSL can be defined through UML profiles or through a metamodel specified with some metamodeling language (for example MOF). Which way to go here is not clear, though. In section 3.1 we develop some reasoning about this particular problem;

2. as we have already noted, the application domain and the scientific bodies of knowledge influence the entire spectrum of model transformations (from PIM to PSM and from PSM to code). It is obviously imperative that the transformations are proved correct for some definition of correctness. As the proof of correctness may be considerably onerous we are interested in taming its complexity to the maximum possible extent. We must therefore understand what factors most influence the relevant costs: in section 3.2 we single out three such factors, which concern: (i) the number of views defined on the PIM; (ii) the incremental construction of the PIM; and (iii) the number of metamodels used to support views and models in both PIM and PSM.

3.1 PROBLEM ONE: HOW TO DEFINE DSL

DSL form an important foundation for MDE, because they permit to express concepts rooted in application domains and in scientific bodies of knowledge. Deciding how DSL are defined is a difficult design decision, which effects the very construction of MDE infrastructure. There are two main ways to define a DSL:

1 to specialize the UML 2 metamodel through the definition of a UML profile. In short, a profile is a particular kind of package that contains elements called stereotypes, which can be attached to certain kind of UML entities (e.g. classes) in order to extend their semantics (e.g. this is a Java class). In this thesis we will consider UML 2.2;

2 to create a new metamodel from scratch using MOF or other metamodeling languages.

A thorough comparison between UML profiles and metamodeling can be found in [15], in which metamodeling is rated more powerful but also less supported by tools than profiles. We think that the answer is not so unequivocal and that there are other considerations that are worth making:

- how UML profiles are defined in UML 2;
- whether, after the introduction of MDA, MOF is favored over UML for defining DSL;
- in which way tools allow to create DSL.

Let us discuss these issues in isolation.

3.1.1 *Definition of UML profiles*

The way profiles are defined in UML 2 seems a little fuzzy to us. In our opinion, this fuzziness stems from the following factors:

- profiles are described in both UML Infrastructure [16], which describes the foundation of the UML metamodel, and UML Superstructure [17], which describes the rest of the UML metamodel. The respective descriptions are almost the same (except for the graphical notation added in Superstructure) and we see no value in the duplication. Indeed, this way it's not clear whether profiles are part of the core of UML or not;
- the description of profiles is poor. To us the specification proved very difficult to parse, in that it is written in a confused way and it is not well structured;
- at first glance, profile have a visibility mechanism to target only a subset of the UML metamodel, so to create a language that is smaller than UML and thus more manageable. In particular to this purpose the Profile metaclass has two associations: (i) metamodelReference, which permits to refer part of the UML metamodel; and (ii) metaclassReference, which permits to specify directly metaclasses extended by the profile. In fact, it seems to us that they don't accomplish that goal:
  - metamodelReference is limited to the compliance levels (subsets of the UML metamodel) and the packages defined in UML – we can still import unwanted metaclasses;
  - metaclassReference seems to address the latter concern, but its use can be quite laborious. Indeed if we want to leave out a few metaclasses we have to use metaclassReference for each of the remaining metaclasses, which can be numerous;
  - last but not the least, the entire visibility mechanism seems a futile exercise, since in the specification after its description we find the following clause:

    > "The filtering rules defined at the profile level are, in essence, merely a suggestion to modeling tools on what to do when a profile is applied to a model."

> In our opinion, this means that all the mechanism is optional (for instance, we didn't find any such mechanism in Papyrus [18]) and in fact in the profile definition we are referencing the entire UML metamodel, which may well be an unwelcome burden.

In summary, we contend that the current definition of UML profiles has flaws that hamper their comprehension and their use.

3.1.2 *MOF and UML*

It is certainly worth investigating and clarifying the relationship between MOF and UML, in order to understand which is best at creating DSL.

This might at first seem an easy question. Indeed, there has been a time when UML and its profile mechanism were promoted as the sole standard way to define DSL. When the MDA initiative proposed MOF as the standard base to specify the metamodels for modeling software systems, however, the opinions changed as some had reasons to prefer MOF to UML for specifying DSL.

In fact the situation isn't clear either way, unfortunately. To begin with, MOF and UML were actually born together: both the Request For Proposals (RFP) for MOF [19] and the one for UML [20] were issued simultaneously in 1996, though in different contexts – MOF for CORBA, UML for modeling. OMG soon acknowledged that UML could be metamodeled with MOF and it was wise to use the same core constructs for both languages. Indeed the RFP for UML 1 required to furnish a mapping between MOF and UML constructs and the UML 1.1 proposal [21] has it.

The specifications of both MOF 2 [22] and UML 2 Infrastructure [16] bring this integration to maturity. As required by their RFP ([23] and [24] respectively), considerable effort has been devoted to develop a core suitable for both languages, which contains (i) basic data types, (ii) abstract concepts needed in the definition of metamodels and (iii) concrete constructs related to object oriented modeling. This approach has at least two advantages according to [22]:

- it avoids the creation of yet another modeling language. UML is widely known and moreover its base concepts are apt to model modeling languages;
- it eases the creation of metamodeling tools through the adaption of existing UML tools. In a broader sense, UML tools are already capable of metamodeling because they already got the necessary language core.

Put otherwise, MOF is a formal way to say that UML is capable of metamodeling. MOF and UML are thus equally important: UML provides the modeling notation and MOF adds facilities useful for metamodel management. In some sense, it can be seen as an incarnation of the UML 2 proposal endorsed by the Distributed Systems Technology Center [25].

3.1.3 *Tools for creating DSL*

So far we have only considered methodological arguments, but it is useful to also have a look at tools for creating DSL. In theory the methodology should drive the technology: but it often happens that tools favor some methodology and make more expensive others. Thus we should look at what way the existing tools let us create DSL, to gage which way they lean, whether for UML profiles or for metamodels.

We first searched for tools suited for creating DSL in the research literature, on Model-Driven Development Tool Implementers Forum home page (`http://www.dsmforum.org/events/MDD-TIF07/`) and on Johan den Haan's microblog on MDE (`http://twitter.com/ModelDriven`). The tools we found include: Eclipse EMF [26]; MetaCase MetaEdit+ [27]; Vanderbilt University Generic Modeling Environment [28]; MOFLON [29]; JetBrains Meta Programming Systems [30]; Microsoft DSL Tools [31] and Microsoft "Oslo" [32]. All these tools use metamodeling in order to specify new languages.

On the other hand, OMG has defined several DSL using profiles and these DSL are supported by industry. For example, MARTE (Modeling and Analysis of Real-Time and Embedded systems) [33] is a UML profile for real-time and embedded systems and Systems Modeling Language (SysML) [34] is a UML profile designed for systems engineering. Both profiles are endorsed by several vendors and tools, for example Artisan Studio by Artisan Software Tools [35] (support for MARTE is only planned) and Papyrus [18].

In conclusion, both metamodeling and UML profiles feature a range of supporting tools.

3.1.4 *Additional considerations*

At this point we would like to inject some additional observations:

- metamodeling gives the user complete freedom over modeling concepts. Conversely, UML profiles only permit additions to the UML metamodel (in the light of our previous criticisms);
- as noted in [15], one distinct drawback of metamodeling is that, after the definition of the DSL, we should expend considerable effort in the realization of companion tools, like graphical editors. Conversely, UML profiles leverage on existing UML tools;
- we must consider the longevity of standards and tools. UML and MOF are clearly well endorsed and this situation is likely to continue. On the other hand, metamodeling tools typically don't use MOF but employ non standard metametamodels, the longevity of which is difficult to assess.

Despite all the arguments we have presented, it's not clear whether to choose UML profiles and metamodeling for the definition of DSL.

3.2 PROBLEM TWO: PROVING THE CORRECTNESS OF TRANSFORMATIONS

The transformations from PIM to PSM and from PSM to code are obviously central to the essence of our application of the MDE paradigm.

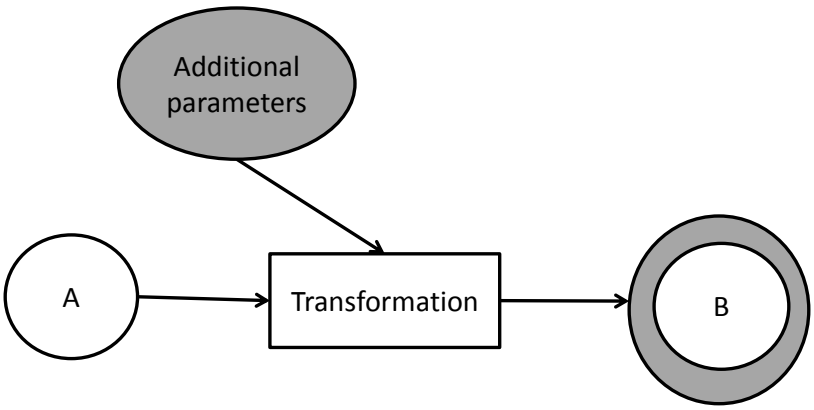

Figure 3.1: Visual depiction of our definition of correctness of transformation. The target model B should contain all the semantics of the source model A and the additional semantics added (usually as parameters of the transformation) should not contradict the semantics established by A.

They turn the user model into a concrete implementation in accord with choices and directives that reflect the application domain culture and legacy as well as methods defined in the scientific body of knowledge. We must therefore ensure that these trasformations are provably correct.

We say that a transformation from a model A to a model B is *correct* if (figure 3.1):

1. everything that holds in A holds also in B. In other words, the semantics of B must contain the semantics of A;
2. things stated in B should not deny things stated in A. In other words, the semantics we add to B should not contradict the semantics contained in A. Note that the trasformation from PSM to code must not add any semantics.

Carrying out such a proof of correctness is a costly endeavor: we must therefore keep its complexity low and its chances of success high. In this respect we wonder what factors impact it most. We focus our attention on three specific factors, which we suspect to play a central role:

- the necessity of specifying large and complex PIM using several views (the word "view" will be introduced shortly);
- the use of incremental PIM construction, which produces multiple intermediate PSM representations;
- the number of metamodels used in the development environment that underlie PIM views and intermediate PSM.

We discuss each of these factors in isolation below.

3.2.1 *Creating PIM from concern specific views*

Most real-world software systems are too complex to be specified with a single PIM – creation, comprehension and evolution may become

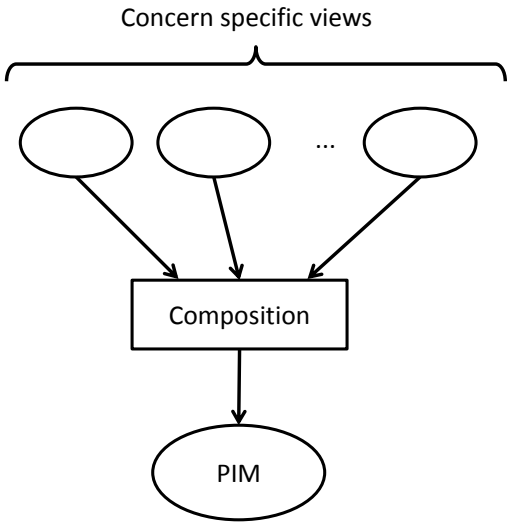

Figure 3.2: PIM does not necessarily consist of a single view, but it may be composed of several concern specific views, each of them represents a particular concern of the system – for example functional units or deployment of tasks on processors. When we compose the semantics of the views we should obtain a single model representing the system under study.

overwhelmingly difficult, because we cram too many things into it. We should therefore rather describe it with several models. In this regard, IEEE P42010/D6 [36] (descendant of IEEE 1471 [37]) advocates that the description of the architecture of a system should be made up of a series of views (figure 3.2), each of which conforms to a viewpoint that establishes the concerns of interest (e.g. deployment or functionality) and the ways to address them (e.g. languages and metamodels). This is further acknowledged by SysML [34] that, with reference to IEEE 1471, supports the concepts of views and viewpoints. Thus PIM should be described by means of views. The more views we admit, the easier and the more accurate the modeling of the overall system.

In order to ensure that the views form in effect a single and coherent system, they should be:

- *composable*: their semantics should not superimpose, or if this happens they should agree on the overlaps;
- *compositional*: there must exist a systematic way to assemble the semantics of concern specific views to obtain the semantics of the PIM as a whole.

In order to trigger the transformation from PIM to PSM, we should prove that the views are composable, carefully minding any overlaps in their semantics. The more views we have, however, the more proofs of correctness we must produce in the PIM space and the more the effort we must expend, which adds to the cost of proving the correctness of the transformation from PIM to PSM.

The cost of determining whether the views are composable in fact depends on the way these views are constructed. There are two main approaches to it, as described in [36]:

- *synthetic approach*: each view is modeled separately with one or more models and later composed with the others;
- *projective approach*: each view is extracted (without any transformation) from a unique underlying model that describes the entire system. Conversely, changes to a view are propagated back to the model.

In the former approach the demonstration of composability of the views implies to show that models underlying them agree on their semantic overlaps. The cost of this activity grows with the number of models – which can be greater than the number of views – and the number of different metamodels used to define them (more on this in section 3.2.3).

Instead in the latter approach this demonstration is straightforward, since views are derived from a single model and this model must be free of contradictions for it to be considered valid. We have still to ensure that views are correctly derived from the model and changes to views are correctly applied to the model, but overall the effort is smaller.

3.2.2 *Incremental PIM construction*

Up to this point we have tacitly assumed that we should have a completely specified PIM before we can instigate the transformation of it into the PSM. This assumption sounds reasonable of course, but it carries the implication that iterative feedback cycles – that we may need to improve the goodness of fit of the PIM – can only be triggered on full and complete models. This prerequisite however may be frustrating, because the best use of feedback cycles is obtained when they begin as early as meaningful analysis can be carried out.

We contend that it is desirable to allow generation of PSM from an underspecified PIM, in order that we can obtain early feedback and permit to refine, change or commit design decisions as early as possible (figure 3.3). While the PIM may be underspecified, therefore, it may still be sufficiently complete for some transformation to PSM to take place for the specific purpose of some specialized analyses. Under this assumption the construction of the PIM becomes incremental. This requirement however has the following implications:

1) things we may leave unspecified in the PIM cannot be arbitrary, but must be determined by the power of the underpinning theory of analysis (which may do away with some detail information) and/or by the analysis procedure (which may permit the use of user estimates in the place of actual values). In other words, it is the PSM and not the PIM to determine what we can omit from the PIM – with this regard the PIM cannot abstract effectively from the PSM and becomes dependent from the particular execution platform;
2) all the PIM increments can be seen as additions or changes to the same model – although the semantics of these increments can disagree because of the corrections made in response to analysis feedback. On the contrary, the increments of PSM cannot be seen the same way but they should considered as distinct models, since they are generated separately and are tailored for the analyses we run

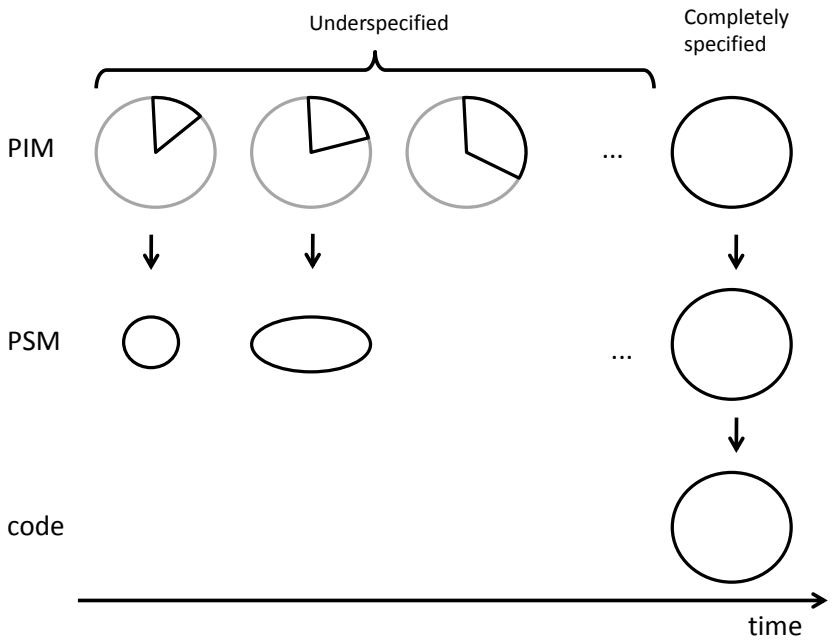

Figure 3.3: PIM can be constructed incrementally in several iterations; over time we may therefore have a string of underspecified PIM. Under certain preconditions an underspecified PIM may generate a PSM apt for some analyses. PSM derived from underspecified PIM are throwaways and cannot be thought of as increment of a single model – only the PSM generated from completely specified PIM should be kept in order to generate the code for the implementation.

on them. This means that every increment is specified with its own metamodel, tailored for the specialized analyses made on top of it. Moreover, except for the PSM corresponding to the completely specified PIM, we can say that all the increments of PSM are "throwaways" – once we have executed analyses on them, they have no further use;

3) the transformation from PIM to PSM becomes more complex – we can in fact regard it as comprised of several transformations, the one for the completely specified PIM (that allows code generation) and the ones for each allowed degree of PIM completeness.

   One important property these transformations should all have is that, given a piece of PIM semantics, all the transformations that address it map it to a piece of PSM that has the same semantics as the one generated by the transformation that targets the completely specified PIM. In other words, we must require that all transformations have a congruent behavior when they operate on the same PIM semantics; if we did not enforce this, then for the various increments of the same PIM we could generate PSM that in fact represent different systems, and the analysis results would in fact be worthless.

   To devise these transformations, it would be best to first define the one that targets the completely specified PIM and next use this as reference to construct the others.

Hence, when we allow incremental PIM construction, the (composite) transformation from PIM to PSM becomes more difficult to prove correct. In fact, to prove that the entire transformation is correct we have to demonstrate that:

1) all the transformations are correct;

2) given a piece of PIM semantics, each transformation maps it to elements in the PSM that have the same semantics as the ones generated by the transformation that works on the completely specified PIM;

3) the preconditions of the composite transformation are correct – that is, we must ensure that for each allowed degree of PIM completeness we trigger the right transformation and only for them.

3.2.3 *Number of metamodels*

We have seen that we may have to deal with a large number of models. Having multiple models implies that we might use more than one language for modeling them and thus we might deal with more than one logical metamodel underneath them. In particular we can have this extreme scenario:

- assuming we are employing the synthetic approach, every model underlying the views in the PIM can have its own metamodel;
- while it is reasonable to assume that every PIM increment uses the same metamodels, we cannot expect this for the generated PSM. Indeed, since in each increment we may perform specialized analyses, we need to express different concerns, for which we may need distinct metamodels.

We talk about *logical metamodel* to suggest that one and the same (mega) physical metamodel may be constructed in a manner that permits multiple logical metamodels to be realized as a specific tailored view of it. For this reason in the following by metamodel we mean logical metamodel.

We saw that the proof of correctness of trasformations requires to handle models' semantics, which is defined through the semantics of their underlying metamodels. We maintain that the number of metamodels to be supported may have a considerable impact on the cost of the proof. If models are specified with different logical metamodels, then before being able to compare their semantics we should perform a *semantic integration* by establishing correspondences between their metamodels' semantics (e.g., this piece of information has the same meaning as that piece of information there; or this relation here is the inverse of that relation there). The cost of this integration seems to grow super linearly with the number of metamodels: if we have n metamodels we have to make a correspondence for each couple of metamodels and then we have $\binom{n}{2} = O(n^2)$ correspondences to make (figure 3.4).

Conversely, if models are specified with the same logical metamodel, then the semantics is defined the same way for all models and it is easier to check for overlaps and contradictions. Moreover, if every piece of

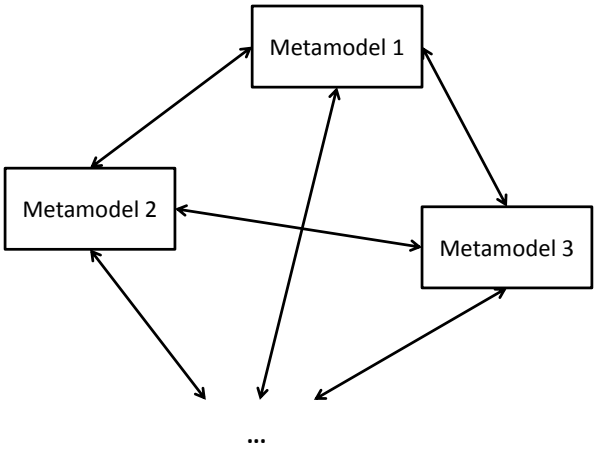

Figure 3.4: Proving the correctness of transformations requires to handle the semantics of models, which in turn is established thought the semantics of their metamodels. To be able to compare the semantics of models, we should first establish correspondences between metamodels' semantics, so to know elements that have the same semantics. If we have n metamodels we have to make a correspondence for each couple of metamodels and then we have $\binom{n}{2} = O(n^2)$ correspondences to make.

syntax is attached to precise predefined semantics, the check becomes syntactic.

In fact, UML actually defines a single logical metamodel with several views defined on top of it (the diagrams).

3.3 SUMMARY

In this chapter we have discussed two problems that are to be faced when we want to leverage application domains and scientific bodies of knowledge in an MDE approach that aims to support the development of correct-by-construction software systems. The problems we posed were the following:

1. how to define modeling languages? Through UML profiles or else through metamodeling? In discussing this question we singled out some important elements:
   - we criticized the way UML profiles are defined and highlighted weak points that hamper the effective use of profiles;
   - we noted that the gain of importance of MOF in the context of MDA didn't undermine the importance of UML as a vehicle for defining languages, but rather recognized that UML is apt for metamodeling;
   - from an informal survey of tools for creating DSL we reported the observation that both approaches seem to be equally supported.

   Nevertheless, neither approach stands out clearly as the right choice.

2. what factors impact most on the complexity of proving model transformations correct? We conjectured that three such factors are:

   - *the number of views that make up the PIM*. We argued that it may be easier to construct the PIM out of multiple, simpler, concern specific views; in that situation however we must show that the views we use to form the PIM are composable, and the cost of this proof is proportional to the number of views. We also noted that it is desirable to have a single model underlying these views;
   - *the incremental construction of PIM*. We insisted that we should enable the generation of PSM from underspecified PIM, in order to facilitate specialized forms of analyses useful for instigating (early) feedback cycles. This approach however incurs a more costly proof of correctness. Indeed the transformation becomes a composite one, for which we have to prove that

     1) the transformations inside it are correct;
     2) given the same piece of PIM semantics, all the transformations that may apply to it must behave the same way as the one that targets the completely specified PIM;
     3) each specific transformation is deployed solely when the degree of underspecification of the PIM allows it.

   - *the number of metamodels used to specify the views of the PIM and the various increments of the PSM*. A proof of correctness requires to deal with semantics; models' semantics are established through metamodels' semantics. Each metamodel defines its semantics in its own way; hence to search for semantic overlaps and spot contradictions we have to make semantic integration between each pair of metamodels. The more metamodels we have, the higher the semantic integration effort.

The discussion of these problems involved only methodological and scientific arguments – first of all we wanted to ensure a sound adoption of MDE in the development process. But technological arguments are equally important – if a solution requires too much effort to be implemented and maintained, it's likely to bring the entire environment to its knees, regardless of its soundness. Therefore in the following chapter we evaluate the feasibility of the approach devised for taming the complexity of correctness proof of transformations using technologies representative of both UML profiles and metamodeling: in this manner we also expect to obtain more elements to decide how to define DSL.

4 TECHNOLOGICAL INVESTIGATION

Mixed conclusions stem from the methodological discussion of the previous chapter. We were able to find some factors that impact on the complexity of proving the correctness of transformations, detailing how to deal with them to render their influence minimal. In particular, we argued that we should employ as few metamodels as possible and that views should be derived without any transformation from a single underlying model. On the contrary, with regard to the definition of DSL, our reasoning highlighted an equivalence between UML profiles and metamodeling, so it might seem we can choose whatever approach we want.

We have already stated that the choice of the way DSL are defined is a very important decision. Indeed almost all the development environment depend on languages:

- a graphical editor is fit for a specific language, in particular it furnishes a tailored graphical notation and facilities to ease the manipulation of models;
- transformations operate on languages defined using a particular metamodeling language;
- the way the language is defined impact on the serialization of models and in particular on the ability to interchange models with third-party tools without losing or corrupting information;
- the management of views is done accordingly to the specific semantics of the language.

Thus we should avoid at all costs approaches to language definition that, although scientifically sound, require great or even herculean effort to support adequately the methodology we devise. Indeed we would risk to undermine the maintainability and longevity of the development environment.

This consideration aids us in the decision between UML profiles and metamodeling approach. Since we have already established part of the methodology, to decide which approach to adopt we will consider technologies representative of UML profiles and metamodeling to evaluate whether, at which degree and at which cost they allow to:

1. define the metamodel. In particular we are interested in the specification of constraints and in the construction of graphical editors;
2. support several views (in particular diagrams) that refer to a single underlying model and are synchronized with it.

In the following sections we present the considered tools, we state more precisely the requirements, we describe thoroughly how each tool adheres to the requirements and at last we assign and discuss synthetic ratings about the suitability of tools for the realization of the development environment.

	Tools	
Approach	Metamodel definition	Realization of graphical editors
Metamodeling	EMF	GMF
UML profiles	MDT UML2	UML2Tools, MDT Papyrus

Table 4.1: Approaches for DSL definition and tools that adhere to them.

4.1 REVIEWED TOOLS

The tools we evaluated are all part of Eclipse, in particular they belong to the Modeling project [38]. All of these run on top of Eclipse 3.5 "Galileo" [39], released on June 24th 2009. A quick summary can be found in table 4.1.

For the metamodeling approach we have chosen Eclipse Modeling Framework (EMF) [26] and Graphical Modeling Framework (GMF) [40]. EMF is a framework that allows the definition of metamodels and the creation of models conforming to these; GMF is a framework for the automated generation of graphical editors for EMF models. Both technologies employ a model-driven approach to obtain the final code. In particular, EMF requires an Ecore model that represents the metamodel; GMF necessitates a series of models that specify the shapes, the creation tools and the relationships between model elements, shapes and tools (in addition to some implementation choices).

EMF and GMF are accompanied by a series of technologies used to implement or complement them (figure 4.1):

- Graphical Editing Framework (GEF) [41]: a mature framework for the construction of graphical editors according to the model-view-controller pattern. It is the base of all graphical capabilities of editors built with GMF;
- EMF.Edit: a subset of EMF that provides the so called *editing domains*, a way to apply changes to EMF models in order to track and undo/redo them;
- EMF Transaction [42]: refines editing domains with transaction semantics, support for concurrent model manipulation and signalling of model changes. It is used by GMF editors to offer robust manipulation of models in spite of concurrent external changes;
- EMF Validation [43] and MDT OCL [44]: the former permits to define constraints for a given metamodel and to evaluate them against models, while the latter implements the Object Constraint Language (OCL) [45] and thus allows the user to specify OCL constraints. They are widely employed by GMF editors to enforce contraints in response to changes to diagrams;
- JET [46]: a model to text transformation languages used to implement and customize model transformations employed by EMF;

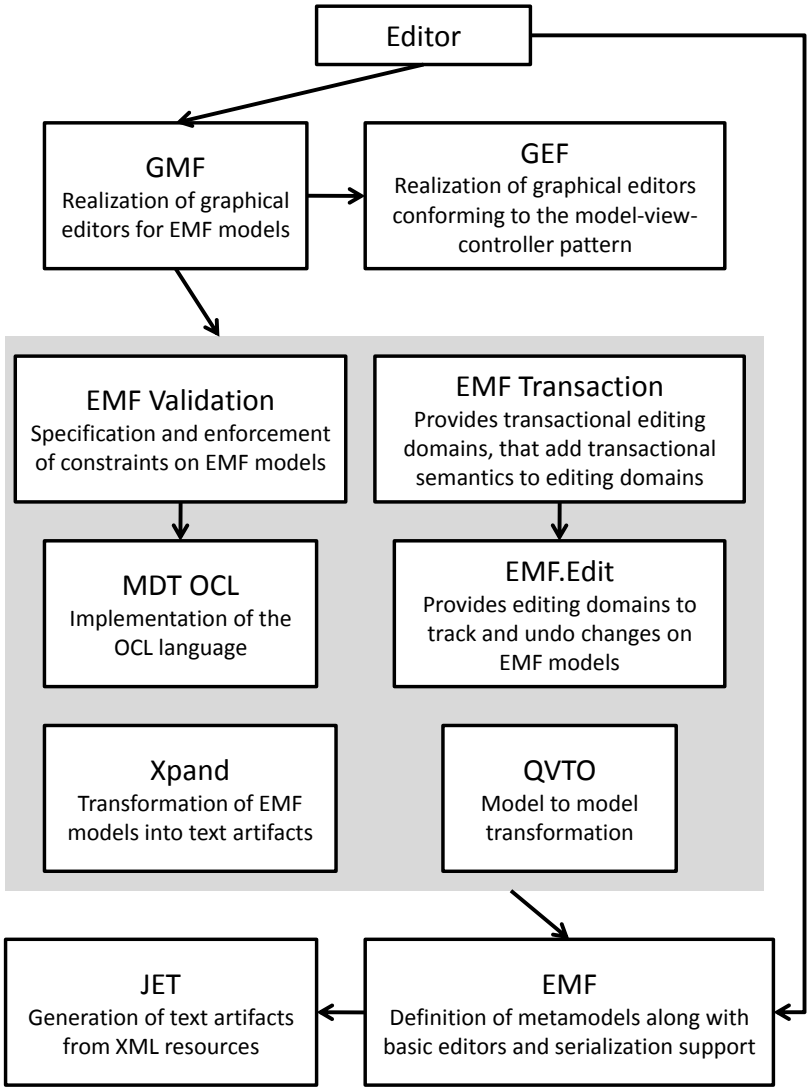

Figure 4.1: Technologies employed by EMF and GMF and relationships of use between them. For the sake of clarity, in the shaded rectangle we have grouped technologies that are used by GMF and use EMF, so to draw only two arrows for them. In section A.3 we detail the exact nature of these relationships.

- Xpand [47]: a model to text transformation languages used to implement and customize model transformations employed by GMF;
- QVTO [48]: a model to model transformation languages used to implement and customize model transformations employed by GMF.

For the profile approach we have chosen MDT UML2 [49] as the implementation of the UML 2 metamodel. In particular, MDT UML2 is aligned with UML 2.2 and it's realized with EMF. With regard to the graphical part, we consider two tools:

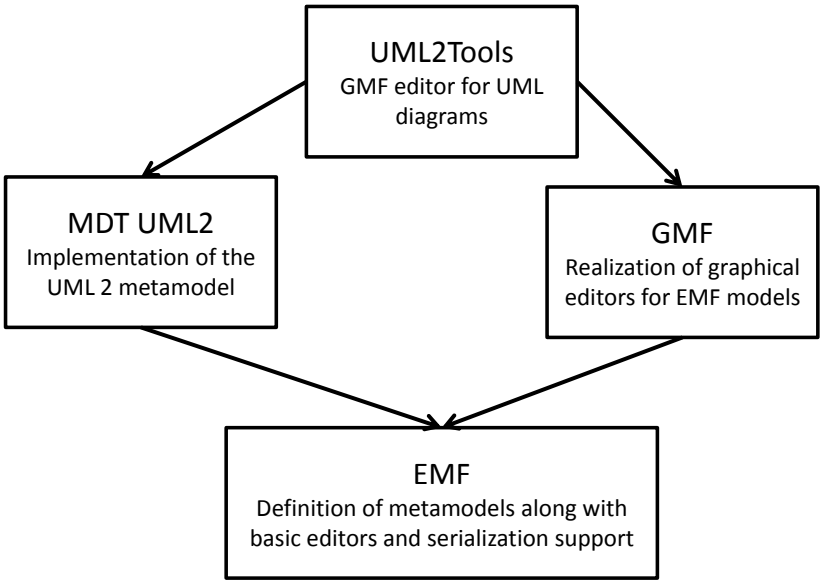

Figure 4.2: Technologies employed by UML2Tools and relationships of use between them. EMF and GMF are used for both construction and execution.

- UML2Tools [50], a series of GMF editors for some of the UML diagrams, devised to be used as building blocks for UML environments;
- MDT Papyrus [51], a modeling environment for the end user with support for UML and SysML [34]. It is being developed from developers of Topcased [52], MOSKitt [53] and Papyrus [18], its direct predecessor developed inside CEA LIST. Let us note that there is no clear naming for this last tool and the one under review – they tend to be called both "Papyrus". For the sake of clarity in this document we stick to the names we have just used.

UML2Tools is still in incubation phase, that is it's still developing processes and communities required by Eclipse. Even so, UML2Tools is used as a starting basis for editors in MOSKitt and TopCased.

Similarly, MDT Papyrus is a project that was accepted by Eclipse on September/October 2008 and it is still under development, without any public release. This means that (i) the program is still rough and (ii) it must be built from sources to be used. We chose anyway to review it since Papyrus, its direct parent, is stable but no more actively developed and supported, and it will be irrational to consider this version. According to [54] it seems that the first public version of MDT Papyrus should be released after summer 2009, that the first stable release will ship with Eclipse 3.6 "Helios" expected for June 23rd 2010 and that there will be capabilities to import models built with the tools it is meant to supersede.

UML2Tools and MDT Papyrus are implemented using EMF and GMF (figure 4.2 and figure 4.3), so the metamodeling and the profile approaches share the same technological basis. We wonder whether EMF and GMF are powerful enough to render viable the metamodelling approach or whether the value added by MDT UML2, UML2Tools

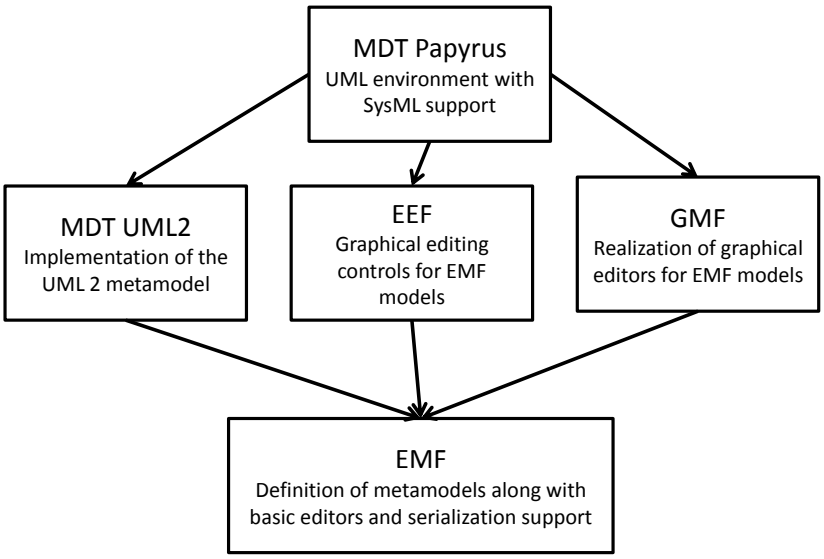

Figure 4.3: Technologies employed by MDT Papyrus and relationships of use between them. EMF and GMF are used for both construction and execution.

and MDT Papyrus can make a difference. For the sake of completeness, let us notice that MDT Papyrus uses also Extended Editing Framework (EEF) [55], which furnishes sophisticated UI elements for editing of EMF models.

We assume that the reader is familiar with the concepts related to plugin development in Eclipse. For further details we refer to [56]. Moreover, we assume the reader is also familiar with GMF: by the way in appendix A we offer a small compendium.

4.2 REQUIREMENTS

Now we state more precisely the requirements that technologies under examination should satisfy in order to be suitable for our needs. For the sake of clarity, we divide the requirements into three categories: those related to DSL definition, those related to view management and those that quantify the required effort.

4.2.1 *DSL definition*

A1. the definition of the metamodel should be rich of options and capabilities.

Rationale For example, some desirable options are rapid prototyping, reuse of existing metamodels and external extensions of an existing metamodel without being forced to redefine it.

A2. it should be possibile to specify a (rich) graphical concrete syntax for the metamodel.

A3. it should be possible to specify constraints that establish whether a given model is valid.

A4. it should be possibile to validate a model while it is being edited.

4.2.2 *View management*

B1. a model can have several views (in particular diagrams) defined on it.

B2. views should be kept synchronized with the underlying model according to some policies that conform to their viewpoints. This synchronization should be done during editing and not only during saving.

Rationale Let us consider a model with two views, a class diagram and a component diagram (in the UML sense). Both account for interfaces. It might seem sufficient to show and synchronize them in both views. Instead it may be better that the component diagram shows and synchronizes only the interfaces used by ports, in order that it contains less clutter.

B3. in a view we should be able to refer an element of the underlying model that was originally created in another view.

Rationale With regard to the previous example, newly created interfaces in the class diagram do not show up in the component diagram. These interfaces can be used by ports only if we permit to reference an existing interface from the component diagram.

4.2.3 *Effort spent on tool construction*

C1. it should be possible to build DSL and graphical editors with a reasonable effort.

C2. it should be possibile to adapt generated artifacts in a way that doesn't require to ditch automatic generation of code.

C3. custom changes to languages and editors should require little knowledge of underlying technologies.

4.3 EMF AND GMF

4.3.1 *DSL definition*

Requirement A1

Using EMF, the metamodel (which in this context indicates only the abstract syntax) is defined using Ecore, a metamodeling language derived from the MOF language. Simply put, Ecore corresponds to the class diagram of UML and it is easy to grasp and use. The metamodel can be specified using the tree editor (figure 4.4), the graphical editor developed inside Ecore Tools [57] (figure 4.5) or the textual syntax provided by EMFatic [58] (figure 4.6).

The code isn't directly generated from the specified metamodel. Instead we should first generate an EMF generator model (genmodel for short, figure 4.7), which decorates the EMF model with additional options about how the code should be generated – for example we could specify different names for the packages or force the compatibility with a given version of Java. Other options will be illustrated throughout this chapter. Once we finished customizing the genmodel,

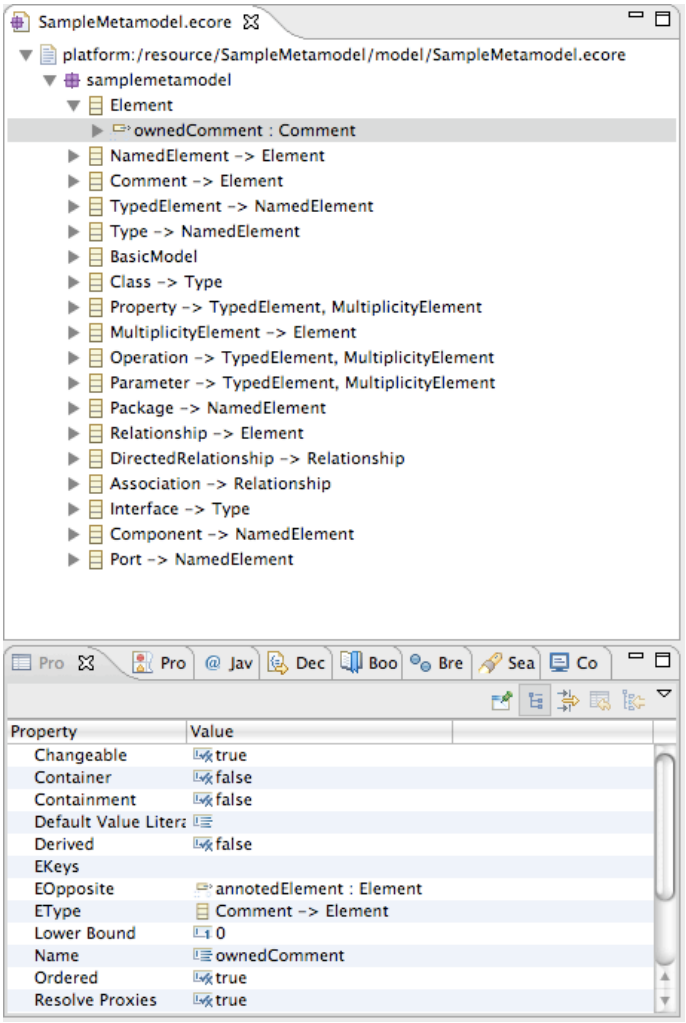

Figure 4.4: The tree editor used to manipulate EMF models.

we can trigger the transformation to code, obtaining a plugin for manipulating from code models conforming to the metamodel and other plugins that provides a basic tree editor (like the one used for Ecore models).

EMF provides some advanced capabilities in the definition of metamodels that are worth mentioning:

- it is possible to reuse elements from an existing metamodel in the definition of a new one – thus we are not forced to start from scratch every time. During the definition of the new metamodel and its associated genmodel we need to reference entirely the existing metamodel, but during the manipulation of models (e.g in the tree editor) we are allowed to create only elements of the reused types;

- EMF allows us to define derived metaclasses for a base metamodel in a new distinct metamodel and to use them in models conforming to the base metamodel [59]. This way we can easily extend existing metamodels for specific purposes, without the need to alter the original definitions. To enable this behaviour, in

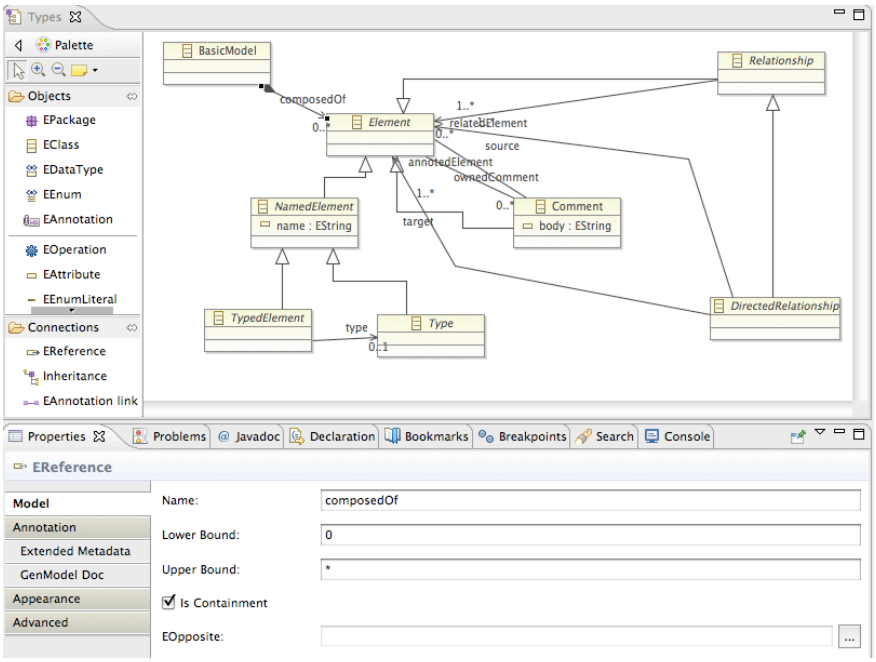

Figure 4.5: The graphical editor contained in Ecore Tools for the manipulation of Ecore models.

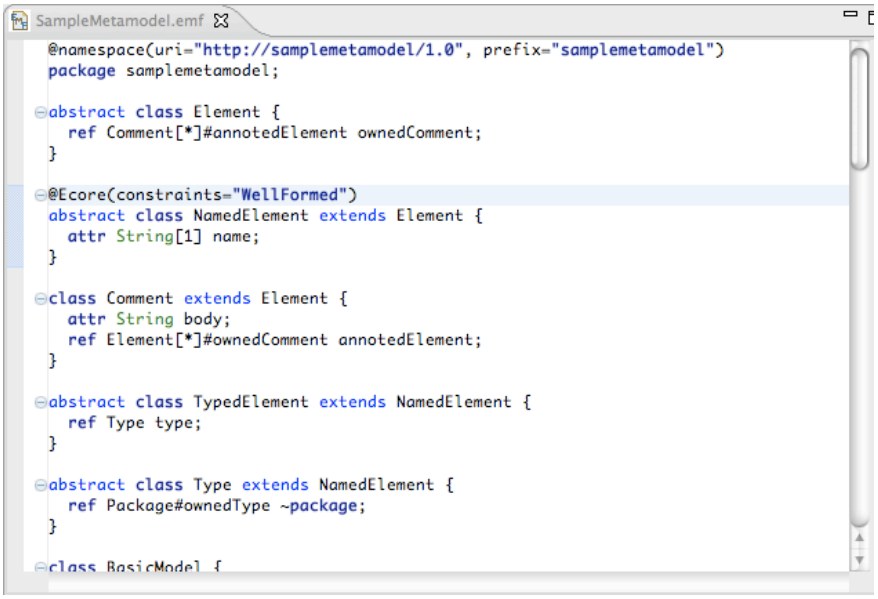

Figure 4.6: An Ecore model specified using the textual syntax provided by EMFatic.

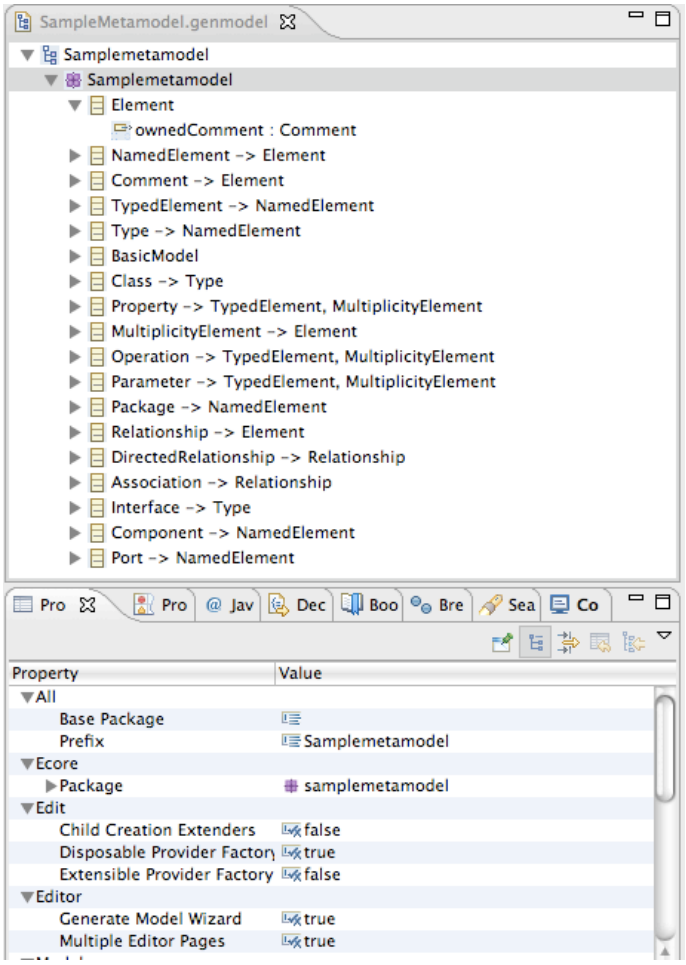

Figure 4.7: A glimpse of a EMF generator model through its tree editor. The tree is the same of figure 4.4, but in the lower part we see properties related to code generation.

the genmodel of the base metamodel we should specify that it allows the creation of new kind of entities (by setting the property "Extensible Provider Factory" to "true") and in the genmodel of the new metamodel we should say that metaclasses could be instantiated in a model conforming to the base metamodel (by setting the property "Child Creation Extenders" to "true"). We will refer to this mechanism as *child creation extenders*.

Requirement A2

The specification of the graphical syntax for an EMF metamodel is done through GMF, in particular using the graphical definition model and the mapping model. The former permits to describe:

- the shapes (for example rectangles, ellipses, even SVG figures and custom polygons);
- the containment boxes, which permit to nest shapes inside other shapes (like packages in UML). In particular, we can specify layout policies (for example all shapes contained in a box are automatically positioned in a stack-like manner);

- the nodes and connections, which have a specific shape. Nodes can also have containment boxes.

Let us remark that we are given lots of options for the construction of the graphical syntax.

The latter model associates nodes and connections defined in the graphical definition model with the elements of the metamodels. This way the graphical definition model can be reused for other metamodels.

The specification of these models can be quite tedious, especially with dense metamodels with several elements. To some relief, GMF provides some wizards to create initial skeletons of these models.

Requirement A3

Although EMF has basic support on validation, we only considered EMF Validation, which is more capable. As the name implies, EMF Validation permits to validate models according to some constraints. Constraints are not specified within the metamodels, but through an extension point provided by EMF Validation. This contributes to flexibility – for example, we can add additional constraints through a new plugin, without touching existing plugins. Out of the box EMF Validation provides two kind of constraints:

- *Java constraints*, that is classes that contain Java code that through model navigation runs checks on the adequacy of the model. In the extension point we should refer to these classes;
- *OCL constraints*. These leverages the implementation furnished by MDT OCL. OCL constraints are directly specified in the extension point, and we find this choice a little awkward for two reasons:
  a. constraints should be inserted in the extension point inside a CDATA section (let us recall that extension points are specified in a XML file called `plugin.xml` and CDATA sections are used to insert data that could be recognized as XML markup). This impedes to modify them using the graphical editor for the extension points and forces the user to manually edit the `plugin.xml` file;
  b. in our opinion, constraints – that is domain-specific knowledge – shouldn't be buried in extension points, that instead contain implementation details on how plugins cooperate – that is computation knowledge. Constraints could be better supplied in external files.

EMF Validation permits to use other languages for the constraints by specifying appropriate language parsers in an appropriate extension point. Moreover, it's possible to retrieve constraints from other sources (for example models or external files) by implementing classes called *constraint providers* and indicating them in an extension point – thus addressing in part our criticisms to specification of OCL constraints.

GMF has some additional facilities to ease the use of EMF Validation. In particular it allows to specify constraints directly in the mapping model, hiding the details on how they should be supplied to EMF Validation.

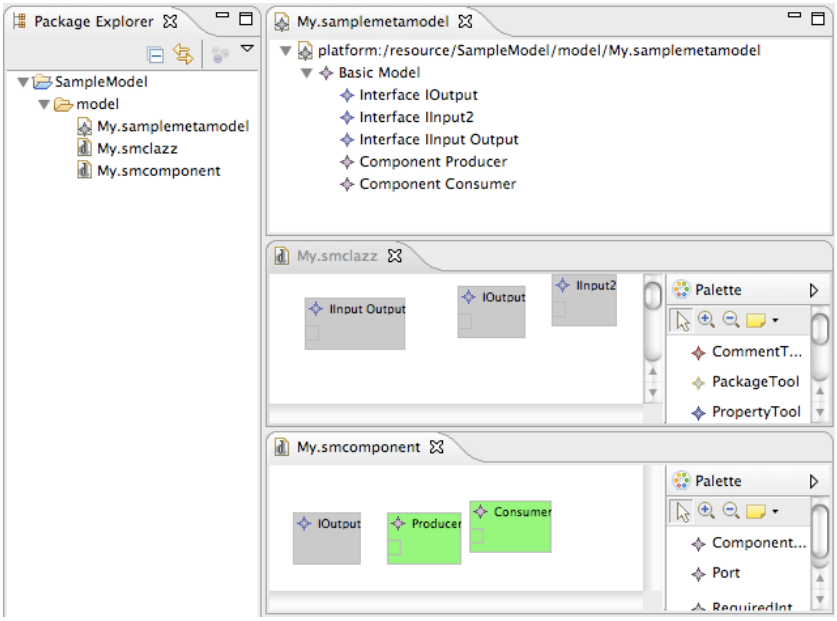

Figure 4.8: An EMF models with two associated diagrams that in fact are views. The top diagrams contains all elements of type Interface, while the bottom diagrams contains the elements of type Component and one element of type Interface.

Requirement A4

EMF Validation provides two types of validation: batch validation done on the entire model and live validation in response to model changes. Within graphical editors generated with GMF live validation is called on user changes and batch validation can be invoked through a menu entry.

4.3.2 *View management*

Requirement B1

GMF does not have an intrinsic notion of views: a graphical editor for a given metamodel is meant to manage a single diagram. However, GMF does not prevent us from defining several editors that target the same metamodel – so each diagram is in fact a view. This is further corroborated by the possibility of creating a new diagram from an existing model, without being forced to create a new dedicated model. To the same model we can then associated several diagrams and thus several views (figure 4.8).

Each diagram is defined in its own file, distinct from the one of the model. If we have too many views, we end up with a lot of files: we found no options or quick recipe to collapse the views into a single file.

For the sake of completeness, we notice that all this discussion does not hold if in the generator model we set the property "Same file for model and diagram" to "true". This way the generated editor will embed the model in the diagram file, and thus we cannot have views.

Requirement B2

When we create different diagrams for the same model, their GMF editors correctly update them if the underlying model changes. Nevertheless, this logic has two drawbacks:

1. if we make a change in a diagram, the change is sensed by other diagrams only when we save it;
2. we can only choose to synchronize all elements handled by diagrams or to not synchronize at all, without any intermediate options.

With regard to the former problem, it is possible to obtain the desired behavior with additional code and changes to the trasformation from generator model to code. Per se the change is not so complex. We already mentioned that GMF graphical editors employ transactional editing domains to manipulate the model. Since every editor has its own editing domain, changes are signalled by the editing domain only to the editor itself and others become aware of them only when we save the model (figure 4.9). To solve this we need share the editing domain among the editors that access the same model (figure 4.10) – EMF Transaction helps in this endeavour. Unfortunately, this scenario is not supported by GMF Tooling; although we can opt to manual change of code, it is better to modify the transformation and this requires some effort. For the implementation details we refer the reader to [60]. In this document we'd like to point out some consequences of this solution:

- when we modify a diagram of a model, all editors that are open on that model become dirty (i.e. they signal their contents are not saved to disk). Moreover, undo/redo stack is shared among editors, since they share the editing domain;
- editors cannot sense changes made to the model with other tools – this is due to the interaction of editing domains with the component that senses changes to the model file. This is not a big problem, since in our intent the user is supposed to use the graphical editors to edit the model.

With regard to the latter problem related to the synchronization policy, this can be solved by coding and plugging into editors an appropriate policy, leveraging GEF and GMF infrastructure. We dwell on the implementation of this policy, that requires appendix A to be fully understood.

In GMF, when we issue a request to create or manipulate a graphical element (which represents an element of the domain), we ask the interested edit part what commands should be executed on both the semantic and notational model (that is, the diagram). This is the right place to check whether the notational model is synchronized with the semantic one and thus generate commands that restore the alignment between the two models. In particular, the edit policy responsible for issuing these commands is referred as the *canonical edit policy* and it's generated automatically by GMF. All we have to do is replace this policy with a different one: we can change directly the code of the edit policy or we can create a new edit policy and, through the use of the dedicated extension point, add it to the edit part while disabling the unneeded one. The canonical edit policy is needed by every shape that

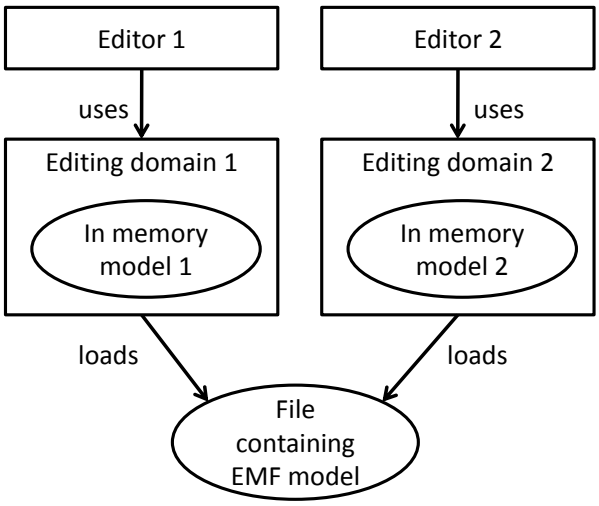

Figure 4.9: Graphical editors generated with GMF has their own editing domains. If two editors open the same model file, in fact the model will be loaded in memory twice. Changes to models done through editors are made on the respective editing domain and thus are not sensed by the other domain.

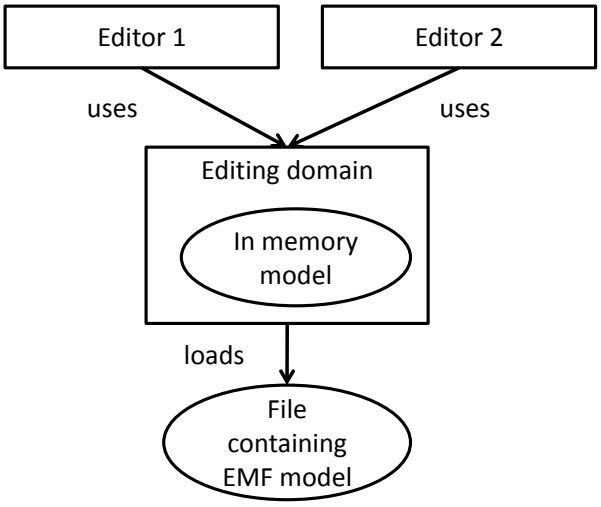

Figure 4.10: The solution proposed in [60] lets the graphical editors share the editing domain when they open the same model file. This way changes made in one editor are sensed by the others.

can contain other shapes – like the entire diagram or the part of the class figure that shows operations and attributes.

We note that the synchronization policy relies heavily on the way references to elements of the semantic model, needed in the notational model, are constructed. EMF models are serialized in XML or XMI format, and references to elements use a syntax similar to the one of XPath. We should avoid *fragment paths* that rely on element position inside its containers, which can be fooled by element reordering or element deletion followed by element insertion. Instead we should use fragment path that relies on keys inside a given container, intrinsic ID (an attribute declared by users as key for all the elements of that kind in a model) or extrinsic ID (an additional key added to all elements, assigned automatically by EMF and guaranteed to be unique). In particular, we tried out extrinsic ID and we verified that, thanks to the use of UUID to identify elements, this approach is immune to the problems mentioned above.

In conclusion, with with the help of some extra ad-hoc code, this requirement is fulfilled by EMF and GMF.

Requirement B3

To insert a notational element for an already existing semantic element, we have two ways:

- using shortcuts, a mechanism provided by GMF to insert references to elements in current and external models (even with different metamodels), through selection dialogs and drag and drop from outlines of the models (which can be found in the Project Explorer view). The mechanism works quite well, although (i) we

incurred some hiccups with selection dialogs and (ii) documentation and tutorials omit to mention a fundamental step for the generation of this feature (the only clue we found to it was [61]);

- programming a command (for example a menu item) that adds a notational element for an existing semantic element. We note that there is some infrastructure to do this but:
  1. it is not well documented. Even looking directly at the source code of GMF it's difficult to understand how to employ the infrastructure, since it is mainly used by creation tools in the palette to generate both notational and semantics elements;
  2. it is a bit rough and not complete. Perhaps it is scheduled for completion in a future release or it is regarded as not important.

4.3.3 *Effort spent on tool construction*

Requirement C1

The construction of metamodel with EMF is cheap – the entire tool chain is robust, easy to use and with good documentation, although we had some difficulties in finding material for advanced topics like resources, child creation extenders and extrinsic ID. Instead building the graphical editor requires considerable effort for several reasons:

- the architecture of GMF is wide and complex, in particular the Runtime part. This complexity stems from the need for extensibility and the nature of the tasks the graphical environment should carry out (graphical editing of figures, management of containment boxes, model manipulation, model validation, definition of notational elements, ...);
- the lack of properly organized documentation, which makes it even more difficult to understand the architecture. The help system contains only outdated articles about the Runtime part, which don't take into account the relationship with the Tooling part. The wiki has several articles and collection of FAQ, but they tend to be outdated, not structured and unclear. There are several presentations from the various editions of EclipseCon conferences (`http://www.eclipsecon.org`), but they tend to have the same material. The newsgroup has a wealth of information, but it's difficult to find if we don't use the right search keywords. Moreover, as of August 20th 2009 web access to newsgroup is clumsy, although we should note that a more robust system is being developed (`http://www.eclipse.org/forums/`). Some useful information is even buried in the Bugzilla system (`https://bugs.eclipse.org/bugs/`).

The lack of documentation for both EMF and GMF forced us to consult the implementations of GMF, MDT UML2 and UML2Tools to understand how to use some capabilities.

Requirement C2

EMF supports direct editing of the generated model code: if the code generator finds a class or a method without any `@generated` javadoc tag

or with the `@generated NOT` javadoc tag, it does not remove or substitute it, preserving custom changes. With regard to methods, EMF can exhibit a sophisticated behaviour: indeed we can customize a method generated by EMF (using the above-mentioned manner) and at the same time ask EMF to generate the original method with a different name, in order to be able to use it in the customized one. Moreover, EMF offers the possibility to modify the trasformation from genmodel to code by indicating a folder with JET templates for the new transformation.

GMF does not accomodate well the direct change of generated code. Instead we have two way to customize graphical editors:

1. the first way to influence the graphical editor is to use the extension points furnished by the Runtime part, which target specific mechanisms. Through extension points we can for example alter the edit policies for specific edit parts or add supplementary notational elements. The use of the extension points hides in part the complexity of GMF and of the generated code. We have also the advantage that additions can be factored in separate plugins, achieving great flexibility;

2. for aspects extension points don't cover, we can customize the transformations from mapping model to generator model and from generator model to code in order to obtain custom implementations. In particular, the latter transformation can be customized in an aspect oriented way, without the need to specify the entire transformation, and the generator model can be extended to allow the specification of additional parameters.

Requirement C3

While EMF is rather self-contained, GMF employs several technologies: GEF for the graphical part, EMF Transaction for model changes and notification, EMF Validation for constraint check, Xpand and QVTO for the implementation of transformations. This means that in order to master the creation of graphical editors the user should get acquainted with the concepts of all these technologies (in particular GEF), in addition to the effort already required to understand GMF itself. This means that we must spend a lot of time only to study properly the technologies and in this period we get little results. By the way, once the user understands the framework and how to gather documentation about it, we believe it becomes quite easy to generate and customize the editor, although the effort is still quite high.

4.4 UML2TOOLS

4.4.1 *DSL definition*

Requirement A1

MDT UML2 offers two ways to define UML profiles, which differ in effort and capabilities:

- *dynamic definition*: we model the profile, next we create an Ecore representation (operation named as *definition*) to be able to apply it to UML models. This representation is embedded into the

profile itself. The profile can be registered in an extension point in order to ease its retrieval within UML tools – this avoids the user to manually search for and load the desired profile. This approach has the advantage of being extremely quick while we lose some control (more on that later).

- *static definition*: the workflow becomes similar to the one for EMF. First we define the profile, we decorate it with stereotypes that guide code generation, we derive a genmodel, we change its options and at last we generate the code for the profile. Code generation within UML 2 seems a little bit more powerful than Ecore code generation: the generation of the genmodel can be influenced with several parameters and in the genmodel we have additional options. As for dynamic profiles, static profiles can be registered in a dedicated extension point so UML editors can retrieve them easily.

  While it is quite laborious with regard to dynamic definition, static definition has several good points:

  - it's easier to manipulate stereotypes from code. In particular, stereotypes are mapped as proper EClasses (the equivalent of UML classes in Ecore) thus the retrieval and modification of their properties are very easy. Meanwhile, in the dynamic definition we have no such EClasses and to modify a given property we are forced to use methods of UML elements that require us to specify its name and the stereotype it belongs to;
  - we can have more control on versioning. Indeed in dynamic definition every change in profile causes the creation of a new Ecore metamodel, while with static definition we have more control on whether to update a given profile or create a new one;
  - we can provide code for operations and derived properties. This is just not possible with dynamic definition.

Static profiles seem to be the preferred approach for defining profiles in MDT UML2. Nevertheless, there some hiccups to be aware of:

- we have noticed that playing naively with the available options can lead to not functioning profiles. For instance, we followed the tutorial contained in the update site [62] (associated with [63]) and we weren't able to compile the generated code. We searched the newsgroup and discovered [64] that this was caused by a misconfigured option in the wizard for the generation of the genmodel, that lead to the generation of invalid code (to be precise, the option "Camel Case Name" was not set to "Ignore");
- in theory, if we need to regenerate the genmodel with different options, the transformation should take into account an already existing genmodel in order to preserve user-defined parameters. In fact, we have experienced that this mechanism is ineffective, since it maintains the old settings regardless of the new ones. Thus we are forced to delete the old genmodel, to regenerate it from scratch and to specify again custom settings.

MDT UML2 includes some welcome facilities like the package merge mechanism and supports child creation extenders to create, instead of stereotypes, new metaclasses in the UML metamodel.

To be precise, so far we have talked about mechanisms offered by MDT UML2 through its tree editor. This editor is functional but is not apt for production-oriented use. UML2Tools has a graphical profile editor that permits to define visually stereotypes, but it does not allow the definition of the profile as described before, forcing the user to use the tree editor.

Requirement A2

UML2Tools furnishes graphical editors for UML for class, deployment, composite, component, state machine and activity diagrams. There is also an editor for sequence diagrams, but it is not fully functional; the editor for timing diagrams is present in CVS but not advertised. There is no editor for communication diagrams.

The graphical editors permit to apply profiles and stereotypes to packages and elements respectively, with support for the display of custom icons for stereotypes – although we weren't able to show icons for stereotypes contained in the MARTE beta 2 profile shipped with Papyrus. We noticed that sometimes this mechanism doesn't work – we experienced this with some models that used the above mentioned MARTE profile. In this regard, the tree editor shipped with MDT UML2 is more robust in profile and stereotype application – but obviously it does not support the visualization of custom icons.

Requirement A3

OCL constraints can be specified directly in the profile and, if we opt for static profile definition, they can be automatically enforced in generated code. Obviously, this leverages MDT OCL. Constraint specification can be done through both tree editor and graphical editor.

Requirement A4

Live validation can be applied also to UML profiles, given that UML is metamodeled using Ecore and thus we can use EMF Validation. It is worth noting that, according to [65], the application of a stereotype to an element is not fired as a change to the element to which is applied but as a change of the model, requiring some extra logic to relate this change to the element.

4.4.2 *View management*

Requirement B1

Since UML2Tools is generated using GMF, we can create several diagrams that refer to the same UML model. All the discussion made for EMF and GMF can be repeated here.

Requirement B2

Within the Galileo release train the synchronization mechanism for the notational and semantic model has been refined in order to let the

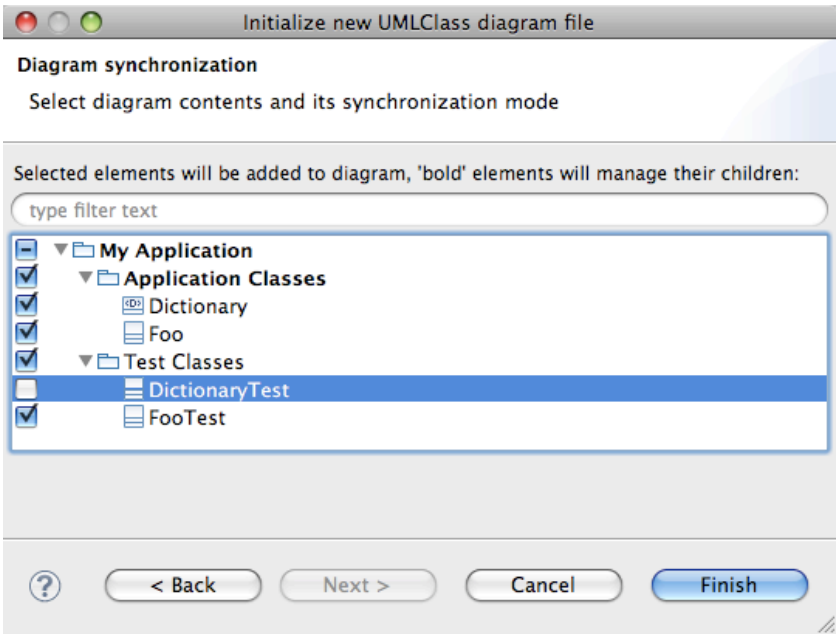

Figure 4.11: Dialog with synchronization settings shown during the creation of an UML class diagram in UML2Tools. Selected element are inserted in the diagram; content of the bold package will be automatically synchronized, while the plain package will not.

user decide which elements to show on the diagram and which containers should have automatic synchronization (figure 4.11). However, synchronization seems not to work with properties and attributes in classes.

Unfortunately, UML2Tools diagrams that use the same UML model are not synchronized during editing but only during saving.

Requirement B3

We can use the synchronization mechanism to select which entities to show in a diagram or in alternative we can opt for the shortcut mechanism.

4.4.3 *Effort spent on tool construction*

Requirement C1

The definition of profiles requires an effort that is the same or less that the effort required in EMF. In addition we already have quite capable graphical editors with full support for profiles, ready to be customized.

Requirement C2

We can take the code of UML2Tools and customize it using the techniques explained for EMF and GMF: thus we can customize them without ditching code generation.

Requirement C3

UML2Tools is realized with GMF and thereby its customization requires to have a good knowledge of GMF and related technologies, and we have already discussed the needed effort. We were expecting to have a simpler mechanism to customize and adapt UML2Tools than the ones supplied by GMF: we soon realized that such mechanism is useless, since GMF is already apt for these changes and further abstraction is not required.

4.5 MDT PAPYRUS

4.5.1 *DSL definition*

Requirement A1

Since MDT Papyrus is based on MDT UML2, all the things stated about mechanisms for profile definition hold here. We have only to see how the tool supports profiles.

As of August 20th 2009, MDT Papyrus is not able to create profiles but it is able to apply them to models. We observe that great care is put on the stereotype application mechanism, as [66] testifies. A minor hiccup we found is that the profiles registered in the extension points of MDT UML2 are not detected in MDT Papyrus; instead, the tool consults another proprietary extension point. In our opinion this behavior has little sense. Anyway, similarly to what happens in the tree editor of MDT UML2 and in UML2Tools, we can point to a profile defined in a project of the workspace.

Requirement A2

As of August 20th 2009, MDT Papyrus has not implemented all the UML diagrams. The effort of the development is concentrated on the class diagram, although we can also find rough versions of sequence, use case, activity and state machine diagrams.

We already said that profiles and stereotypes can be applied, at least in class diagrams. Custom icons for stereotypes should be supported: we found options to control the graphical depiction of stereotypes (textual, graphical, both or shape), but actually only the textual form works.

Requirements A3 and A4

We already said that MDT UML2 supports constraint specification and enforcement. With regard to MDT Papyrus, it is difficult to establish to which degree it will support constraint specification and enforcement in profiles.

4.5.2 *View management*

Requirement B1

Also with this tool a single UML model can have several diagrams defined on it. While with GMF and UML2Tools each diagrams has its own file, in MDT Papyrus all diagrams are grouped in a single file.

This way a model has only one diagram file, resulting in less file clutter in the project.

Requirements B2 and B3

The requirements about view synchronization and references to model elements cannot be verified because as of August 20th 2009 development is focused on each single diagram (in particular the class diagram). To have an idea of the capabilities of the final version, we briefly examined MOSKitt 1.0.0 RC1 and Topcased 2.5 and we found interesting features we hope will be ported to MDT Papyrus, in particular: (i) we can drop elements from the outline of model (shown in the Project Explorer) into the diagrams; (ii) sequence diagrams correctly recognizes and proposes classifiers on class diagram as types for lifelines; and (iii) views are kept synchronized with the model during editing (although not completely, for example properties and operations of classes are not kept synchronized).

We believe these features and complete support for view synchronization will be present in the final version of MDT Papyrus.

4.5.3 *Effort spent on tool construction*

Requirements C1, C2 and C3

We can repeat the same considerations shown for UML2Tools, given that also MDT Papyrus is based on EMF and GMF. We note that with regard to UML2Tools MDT Papyrus is overall better refined and with more features, despite it is still under development. As an extreme example, the editor of sequence diagrams furnished with UML2Tools is not fully functional, while the one in MDT Papyrus is quite usable.

4.6 EVALUATION

From the considerations we have made so far we want to derive a series of synthetic ratings that enable us to discern which is the most promising technology.

4.6.1 *Assessment of requirement satisfaction*

For each requirement we assign an integer score in the range $[0, \ldots, 9]$, where 0 (zero) means that the requirement is not satisfied, 5 that the requirements is barely satisfied and 9 that the requirement is satisfied fully. The scores for the reviewed tools are reported in table 4.2.

4.6.2 *Obtaining synthetic rating*

For each technology we want to obtain a single mark that indicates its adequacy for the realization of our development environment. The idea to get this mark is to first establish the importance of requirements assigning a weight to each (expressed in percentage) and next to calculate the rating using a weighted average of requirement scores. For the sake of simplicity, we also assign weights to categories, so that weights of requirements are in fact relative to their category. Thus the rating of a tool is calculated as weighted average of category scores,

			MDT Papyrus	
Requirement	EMF/GMF	UML2Tools	Current	Potential
A1	8	8	5	8
A2	7	8	5	8
A3	6	7	6	8
A4	7	7	6	8
B1	6	6	7	7
B2	6	8	2	8
B3	5	7	2	7
C1	5	7	7	7
C2	8	8	8	8
C3	2	3	3	3

Table 4.2: Table with characteristics of analyzed tools. Scores are between 0 and 9.

that in turn are obtained as weighted averages of requirement scores. We finally order the analyzed technology according to the previously computed ratings.

One might argue that the final ratings might be severely affected by the weights assigned to the requirements and the categories. Thus we want to apply different weight assignments in order to compare how they affect the ranking of technologies. We have devised four assignments:

- W1 in table 4.3. Every category has the same weight and within each category requirements have the same weight. Put another way, every requirement has the same importance, leading to a reference assignment;
- W2 in table 4.4. We assigned weights to categories in order to reflect their importance, while the requirements within a category have the same weight. In particular, we regarded as more important the second and third category;
- W3 in table 4.5. Every category has the same weight but within each category requirements have different weights reflecting their importance. In particular, in the first category we assigned a low weight to A4 and a high weight to A2, in the second category we raised the weights of B2 and B3, in the third category we increased the weights of C2 and C3;
- W4 in table 4.6. In this assignment we combine the category weights of W2 with the requirement weights of W3.

4.6.3 *Final ratings*

In table 4.7 and graphically in figure 4.12 we have reported the ratings obtained by EMF/GMF, UML2Tools and MDT Papyrus for each weight assignments. With regard to MDT Papyrus, we also inserted the ratings that we reckon it will achieve when it will exit the development phase.

Requirement	Weight within category	Category weight
A1	25%	33%
A2	25%	
A3	25%	
A4	25%	
B1	33%	34%
B2	34%	
B3	33%	
C1	33%	33%
C2	34%	
C3	33%	

Table 4.3: W1 weights.

Requirement	Weight within category	Category weight
A1	25%	20%
A2	25%	
A3	25%	
A4	25%	
B1	33%	40%
B2	34%	
B3	33%	
C1	33%	40%
C2	34%	
C3	33%	

Table 4.4: W2 weights.

Requirement	Weight within category	Category weight
A1	25%	33%
A2	40%	
A3	25%	
A4	10%	
B1	10%	34%
B2	45%	
B3	45%	
C1	10%	33%
C2	45%	
C3	45%	

Table 4.5: W3 weights.

Requirement	Weight within category	Category weight
A1	25%	20%
A2	40%	
A3	25%	
A4	10%	
B1	10%	40%
B2	45%	
B3	45%	
C1	10%	40%
C2	45%	
C3	45%	

Table 4.6: W4 weights.

	Scores			
			MDT Papyrus	
Weights	EMF/GMF	UML2Tools	Current	Potential
W1	5.680	6.845	5.043	7.122
W2	5.328	6.712	4.968	6.944
W3	5.847	6.888	4.480	7.038
W4	5.620	6.730	4.330	6.840

Table 4.7: Comparison of several way of obtaining final scores of technologies.

We can easily see that the influence of weight assignments on final ratings is not strong enough to change the ordering of the tools, which in every assignment is UML2Tools, EMF/GMF and MDT Papyrus (considering its current state).

MDT Papyrus deserves a remark. It is a promising tool, but its current state renders too difficult to assess its capabilities – and hence its

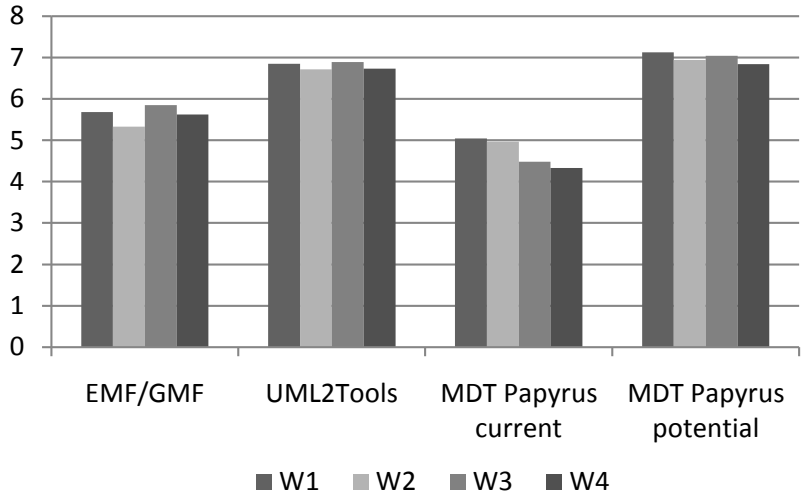

Figure 4.12: Histogram for the final ratings presented in table 4.7.

low rating. Given the remarkable pedigree of its "predecessors" (TopCased, MOSKitt, Papyrus) and the effort put in the development, we believe that it can at least reach the level of UML2Tools.

4.7 CONSIDERATIONS

The ratings we have calculated clearly show that from a technological point of view UML profiles perform better than metamodeling. Interestingly, UML profiles are also the preferred choice by industry, due to their standard nature and the broader (perceived) support they have. While the two approaches are equivalent with regard to the mere definition of languages, we see marked differences in view management and needed effort.

The profile approach takes great advantage of existing UML editors, that usually are full of capabilities developed by long-time experts. This means that we have a good infrastructure that needs few and little adjustments to be incorporated in an MDE development environment.

On the other hand, metamodeling is more flexible but requires to expend considerable effort to support every language we devise for our environment. In the case of EMF and GMF, part of this effort is due to the lack of proper support of quite common features, which requires the user to realize extensive changes and additions – like live synchronization of diagrams using the same model. Even with more capable tools, we believe that we still need significant effort that cannot be avoided – indeed every language has its peculiar characteristics that must be specifically supported. As an informal proof, as of September 10th 2009, the development of class diagrams and related profile support in MDT Papyrus lasted for more than three months – and it's not over yet.

Let us remark that these results are drawn at a given moment in time and things can change in the future with the introduction of new features in considered tools.

We would like to make a final remark. Inside Eclipse there are duplicated UML2 editors, although it was stated that UML2Tools concentrates on automatic editor generation and MDT Papyrus on building the end user experience. It is interesting to see how this duplication will be handled. In our opinion, it might happen than one set will be chosen and the other discontinued. This can be a problem for the people that have chosen the discontinued tools – although they are based on GMF and thus can be maintained by adopters, GMF requires lots of knowledge that hardly a normal user can have.

4.8 SUMMARY

Choosing how to define languages in a MDE development environment should not only based on methodological reasoning, but also on technological considerations. In fact, all the features of the environment are implemented on top of languages: if the way the languages are defined makes the implementation difficult, our environment will become costly to build and maintain.

To this end, in this chapter we compared tools representative of UML profiles and metamodeling in order to evaluate the effort needed to:

1. define the metamodels of languages, with particular attention to graphical syntaxes and constraints;
2. implement views defined on a single model by means of diagrams, a piece of our solution to reduce the complexity of proving the correctness of transformations.

All the tools run on top of Eclipse.

For the metamodeling approach we chose EMF and GMF. The former permits to define metamodels while the latter enables the creation of graphical editors for them. While these technologies are quite mature and full of capabilities with deep support for extensibility, we find important flaws:

- they lack an extensive and organized documentation, in particular with regard to GMF and advanced features of EMF;
- the effort required to implement views is considerable. Contrary to our expectations, GMF does not offer ready-to-use facilities to this end but requires manual modifications.

Tools for UML profiles are more promising. We chose two tools, UML2Tools and MDT Papyrus. Both are based on the MDT UML2 implementation of the metamodel of UML2 and both employ EMF and GMF.

MDT UML2 provides a solid implementation of UML2, in particular it offers several options for the definitions of profiles.

UML2Tools consists of a series of GMF editors for some of UML 2 diagrams. These editor are quite refined, with remarkable support for views – it's even possibile to choose which elements to show in a diagram. UML2Tools is not thought for end users but as a starting point for UML 2 tools.

MDT Papyrus is an undergoing effort to build a full-fledged UML 2 environment, with support for SysML. It has an high potential, with

more polish than UML2Tools: unfortunately it is quite difficult to assess its capabilities because it is under development and it lacks several important features.

To conclude, we found that tools supporting UML profile are more effective that those that support metamodeling, because they incorporate features that ease the development of additional capabilities, while the latter ones are quite low level. This conclusion is even stronger if we consider that in this particular comparison UML tools are built using metamodeling technologies – this means that the support of languages is very important.

5 CONCLUSIONS

Model Driven Engineering (MDE) is a promising approach that permits to develop complex software systems with higher quality and shorter development times than approaches that employ manual coding and programming languages. It is based on two principles:

1. software is specified by means of models that express with clarity requirements and functionalities using the concepts of the application domain. This way we avoid the burden of computation concepts specific to coding and programming languages and furthermore we allow stakeholders, who know well the domain and its intricacies but are perhaps less knowledgeable when it comes to implementation, to directly modify the model. The languages used to create the models are called Domain Specific Languages (DSL), since they are specific to a domain and do not contain any foreign concept;

2. the final implementation of the system is obtained through automated transformations from models. The use of transformations guarantees that the implementation is coherent with the specification and obtained thorough the use of proven solutions and patterns. Moreover, we can change the underlying technology by simply changing the transformations, without need to replace the models.

Nevertheless, MDE should be applied correctly in order to obtain these benefits: if we overlook some aspects we will risk to spend too much effort on the construction and the maintenance the development environment, hampering the effectiveness of MDE.

Thus in this thesis we have discussed several interesting facts we hope to be of assistance in unleashing correctly MDE in the development of software systems, in particular high-integrity and real-time systems. For this purpose, we considered a particular MDE initiative, Model Driven Architecture (MDA), endorsed by the OMG. In this approach the system is first specified using a Platform Independent Model (PIM) which abstracts away from technological aspects and focuses on functionalities; then we choose the execution platform and its parameters and we transform the PIM into a Platform Specific Model (PSM), on which we run static analyses to assess its adherence to requirements. If the results of analysis don't satisfy us, we go back to the PIM; otherwise with a transformation we turn the PSM into the final implementation of the system, ready to be compiled and executed.

Our first contribution highlighted the very nature of application domains. While they are surely composed of specific concepts and problems that require dedicated solutions, they also share concerns with similar domains – consider for example the need to guarantee temporal deadlines in automotive, railway and aerospace domains (to name a few). At first glance it might appear that this duplication is not so troublesome and every application domain can tackle on their own these concerns. Beside the fact we would waste considerable effort in

resolving over and over again the same problems, these concerns are such that they require scientific knowledge, method and discussion to be tackled effectively, especially if we want solutions that are sound and cost effective. Thus application domains should be supported by so called scientific bodies of knowledge, which provide proven solutions to transversal and recurrent concerns.

In our second contribution we presented two problems that stand before us in the construction of an MDE development environment that leverages these two kinds of domains and guarantees correctness-by-construction with high confidence and reasonable effort. These problems are:

1. *the definition of DSL*. These languages should express appropriately the concepts of both application domains and scientific bodies of knowledge. We can choose to implement them with UML profiles or with metamodeling, and it is not clear which way to go;

2. *the complexity of proving the correctness of transformations*. These proofs are important in that they guarantee that output models do not distort the semantics of input models and that both domain-specific and scientific knowledges are correctly applied. Unfortunately, these proof are very expensive. We are thus interested in understanding the factors of the development environment that impact the most on this complexity and the way to lower their influence.

We analyzed and discussed these problems from a scientific point of view, trying to devise some ways to address them or to decide on them. With regards to the latter problem, we were able to isolate some factors that impact on the complexity of proofs:

1. *the number of views that compose the PIM*. A software system should address a conspicuous number of concerns (e.g. security, concurrency, performance, ...) and this is reflected by its complexity and size. We can obviously represent such a system with a single model – but its specification and comprehension will be greatly impaired, since the various concerns tend to be tangled. For this reason software architectures are usually described by means of views – one or more models that represent some specific concerns. Focusing on a few concerns, views are easier to understand and to manipulate. Thus views should be absolutely employed in the specification of PIM.

   However views cannot be arbitrary. Indeed, we have to prove that they are composable, that is that their semantics agree on overlaps, and the effort of this proof grows proportionally with the number of views. The cost of proving composability depends also on the way views are managed. If we opt for the synthetic approach, views are specified with one or more models that are later composed, and therefore the cost is likely to be high due to the number of models and the different employed metamodels. If instead we opt for the projective approach, views are derived without transformations from a single underlying model, so we have only to prove that the model is valid and views are derived correctly;

2. *the incremental construction of PIM*. Realistically, software is not specified completely the first time; instead it is more likely that we specify it piecewise. We would also like to receive early feedback on the

underspecified PIM we construct, in order to validate our design. This is possible if we allow the generation of PSM from this underspecified PIM, in order to run analyses on them and know if our design is apt or not. This capability is desirable but has several important implications on the transformation from PIM to PSM. In fact we need a series of transformations, one for each allowed degree of PIM completeness. These transformations couldn't be devised independently, since they should behave coherently on the same piece of PIM semantics: a reasonable approach is to create the one that targets the completely specified PIM and then derive the ones that work on underspecified PIM. Proving the correctness of all these transformation is clearly more complex, in that we have to show that:

a) each transformation is correct;

b) given a chunk of PIM semantics, all the transformations that may apply to it behave the same way as the one that targets the completely specified PIM;

c) each specific transformation is triggered only for the intended degree of completeness of the PIM.

3. *the number of metamodels used to specify the views of the PIM and the various increments of the PSM*. A proof of correctness of a transformation should be able to compare the semantics of input and output models, and thus it should know the semantics of associated metamodels. If we have one metamodel, we have no problem: but in real approaches we tend to have several metamodels. We have already seen that views of the PIM can refer to different metamodels; moreover if we allow incremental construction of the PIM, each PSM should support different analyses according to the degree of completeness and thus they are likely to have different metamodels.

   Different metamodels define their semantics in different ways. so we need to establish correspondences between the semantics of the metamodels, in order to know how the same semantics is modelled in each of them. This work should be done for each pair of metamodels and thus the effort grows quadratically with the number of metamodels. Having too many metamodels could then result in a considerable endeavour.

The former problem with the definition of DSL revealed itself tougher than we expected. Their innate characteristics aren't enough to decide between metamodelling and UML profiles: the former gives more freedom in the definition of the language but has less tool support, while the reliance of the latter on UML is at the same time a strength – especially with regard to tool support – and a weakness – we can only add to the UML metamodel. We then singled out additional element, hoping that they could guide the decision:

- analyzing the definition of UML profiles, we criticized its poor description and discovered that the mechanism intended for the selection of a subset of the UML metamodel is not fine-grained enough to be useful and moreover is not deemed mandatory for tools to implement – in practice a profile always references the entire UML metamodel. These are important obstacles for the adoption of profiles; on the other hand, there is no affirmed standard

for metamodeling, since every tool has its proprietary metamodeling language, and this affect the longevity of this approach;

- we investigated the relationship between MOF and UML: in the context of MDA the former seems to be preferred to the latter, despite the presence of the profiles mechanism, and we wanted to confirm this impression. A thorough analysis of OMG specifications and requests for proposals has shown how the two standards have indeed the same importance and they are deeply tied to one another, in that they share a language core with concepts apt for metamodeling and class modeling;

- we conducted an informal survey of tools for creating DSL, to see if they lean to a particular approach. We discovered that both approaches seem to be equally supported.

Despite our arguments, there was no clear indication about which approach to pick – so it might seem that we can choose either approach. This answer did not satisfy us, because upon languages we build all the functions of the development environment (editors, transformations, serialization, views, ...) and thereby they determine on the effort needed to implement them. If an approach requires too labor to be built and maintained, our environment might be doomed to failure.

So, as last contribution, we evaluated technologies representatives of UML profile and metamodelling to see whether, to which degree and at which cost they allow us to implement part of our methodology for MDE adoption that we proposed so far. In particular, we drew our attention to two aspects:

1) the definition of the languages, in particular their constraints and graphical syntaxes;

2) the support of views on a model using the projective approach, with the use of a single metamodel and with custom synchronization policies.

All considered tools were part of the Modeling project inside Eclipse. For the metamodeling approach we picked EMF and GMF, two fundamental tools that are used extensively for other tools inside the Modeling project (including the reviewed UML tools). EMF permits to create metamodels of language along with basic tree editors and with serialization support; GMF provides infrastructure and tools for the construction of graphical editors for EMF models. Among their dependencies, we recall EMF Validation, which permits the specification of constraints for EMF models, and GEF, which furnishes the support for the graphical capabilities of GMF. The definition of languages can be carried out quite easily, and EMF provides several options to tailor the generated code to desired needs. The construction of graphical editors is eased by the use of a model-driven approach that avoids repetitive work and at the same time permits extensive customization to support particular behaviours. However, we believe that the effort we spent was excessive: this is due to little support by GMF for common scenarios and the lack of properly organized documentation for both tools.

For the UML profile approach we chose MDT UML2 for the implementation of UML2 metamodels and picked two graphical editors:

UML2Tools and MDT Papyrus. UML2Tools is made up of a series of editors for UML diagrams and it is meant to be reused in UML environments; MDT Papyrus instead is an UML environment devised for the end user, with future support for SysML. All the mentioned tools are based upon EMF and GMF, but despite this dependency they supported far better the aspects of interest. MDT UML2 is rock-solid, with adherence to version 2.2 of UML 2 and extensive support for UML profiles. UML2Tools is provided with a fine-grained synchronization mechanism and graphical support for profiles, although we noted that it is not well polished and it lacks some diagrams. MDT Papyrus showed great potential, given the care showed by the developers, but we should wait for the first public release to assess more precisely its capabilities.

Our evaluation showed that, in this moment, UML profiles permit to implement the chosen approach easily and with less effort: indeed each language should be supported in its own and specific way and building this support for custom languages has a huge cost. This result was further corroborated by the fact that the tools that employ UML profiles are implemented using the ones that employ metamodelling: even with the same technological basis, profiles have a consistent advantage over metamodeling. To conclude, for a sound MDE methodology we should adopt UML profiles, which due to their standard nature are well supported and require less effort to be incorporated into a development environment.

A

ARCHITECTURE OF GMF

GMF architecture is quite wide, in order to support adequately (i) editor generation through transformations and (ii) customization. At first glance it is composed of two major parts, GMF Runtime and GMF Tooling, which we explain in the following sections. In the last section we also clarify the dependencies depicted in picture 4.1.

A.1 GMF RUNTIME

It specializes the infrastructure of Graphical Editing Framework (GEF) [67, 68, 69]. GEF employs the model-view-controller pattern, focusing on the view and the controller and allowing any kind of model (technologically speaking). The view is made up of figures, connections and compartments; the controller is made up of edit parts, which have two important functions:

1. they keep synchronized the view with the model;
2. they translate operations done on the view (e.g. the connection of two figures or the addition of a figure) into operations done on the model. Edit parts delegates the decision of issued commands to specialized components called edit policies: this way the logic that performs this translation is not monolithic, but it broken in more manageable chunks, which can be also be easily replaced and added.

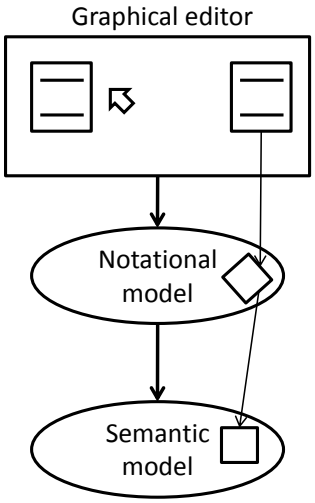

Figure A.1: Models used by a GMF editor. The semantic model contains the data we are interested in; the notational model establishes how to display the entities of the semantic model in the editor.

In the GMF Runtime we find a series of classes designed to make GEF work with EMF models. It is worth noting that GMF editors work with two models [70] (figure A.1):

- the semantic model, which contains the data we want to manipulate;
- the notational model, which specify how entities in the semantic models are displayed in the diagram. The notational model is defined using a generic metamodel and the allowed composite entities are defined through an extension point. Obviously, the notational model contains references to elements of the semantics model.

GMF Runtime provides several extension points in order to customize its behaviours: we can customize which model operations we should execute when the user request an action, we can influence the layout of the shapes, we can modify the palette of tools, we can specify additional edit policies to existing edit parts, and so on.

A.2 GMF TOOLING

Since GMF Runtime is employed in similar way in every editor, GMF Tooling provides facilities to specify an editor at high level and then produces its code through transformations [71, 72] (figure A.2).

Assuming we have already defined the metamodel, we have to specify three additional models:

- the graphical definition model, which specifies all the figures, connections and compartments that we will have on the editor;
- the tool definition model, which contains all the tools available in the editor for creating shapes and connections;
- the mapping model, in which we relate the metamodel, the graphical definition model and the tool definition model, indicating for each domain element: (i) the connections, shapes and containment boxes used to display it; and (ii) the tool used to create it in the graphical editor.

Using a transformation, from the mapping model we obtain a generator model in which we detail several generation details – like configuration options (names of Java packages, ID for plugins and other elements, ...) and behaviours (synchronization policy, validation, ...). Through another transformation, the generator model is used to generate the code for the graphical editor.

GMF Tooling allows some degree of customization [72]:

- the transformation from the mapping model to the generator model can be substituted with a custom one;
- the transformation from the generator model to the code can be customized in an aspect oriented way;
- the metamodel of the generator model can be extended in order to allow the specification of additional parameters, which can be used in the previous transformation.

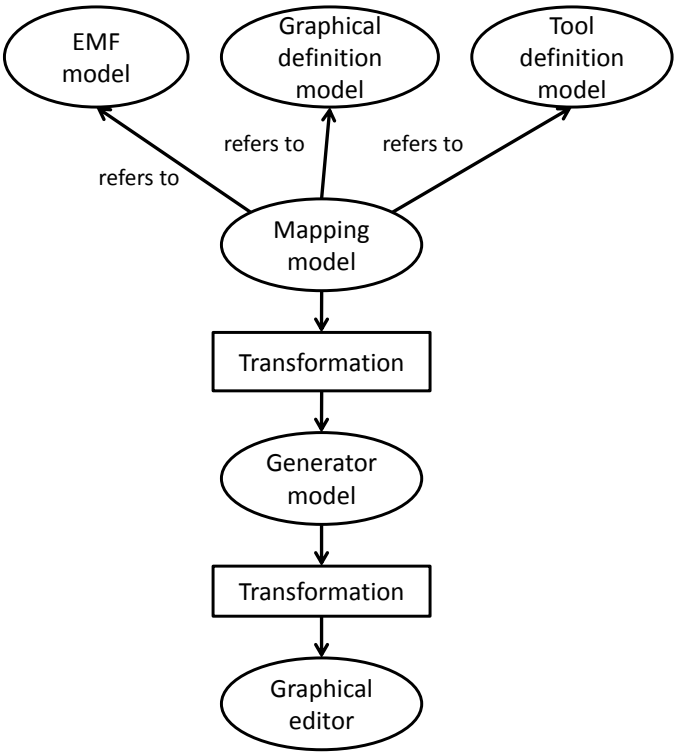

Figure A.2: Workflow employed by GMF Tooling to generate automatically a graphical editor. In addition to the metamodel, we should specify a graphical definition model and a tool definition model. All the before mentioned models are referred by the mapping model, in which we specify for each entity in the metamodel the shape used to show it and the tool used to create it. This mapping model is transformed into a generator model, in which we can specify several implementation details. A final transformation takes the generator model and produces the plugin for the graphical editor.

A.3 DEPENDENCIES

As we described, GMF has a twofold nature: it is used to both generate and execute the graphical editors. At this point the attentive reader will wonder whether the relationships in 4.1 are valid for both Tooling and Runtine or not.

In fact picture 4.1 is simplified in this sense, with the aim to present to the reader all the technologies without excessive burden. Figures A.3 and A.4 portray the exact dependencies of GMF Tooling and GMF Runtime, respectively. Summing up their content, in the tooling part we only employ transformation tools, namely Xpand e QVTO, while in the runtime part we use frameworks and libraries necessary to operate the graphical editor. Moreover, the final graphical editor does not use GMF Tooling, instead it is generated by GMF Tooling.

Also the dependency on EMF is sketched, because most of the time we are not directly dependent on EMF but on the plugins that implement the support for a particular metamodel (e.g. the tooling definition metamodel, the graphical definition metamodel, ...). For the sake of

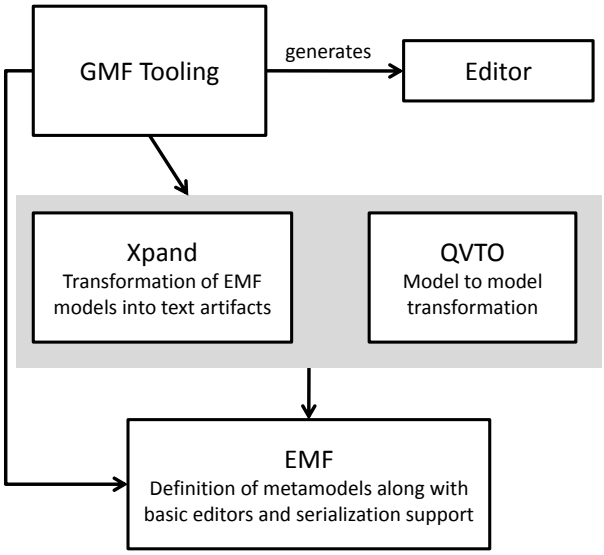

Figure A.3: Technologies employed by GMF Tooling. While in figure 4.1 the editor generically uses GMF, in fact it does not use GMF Tooling, but instead it is generated from GMF Tooling.

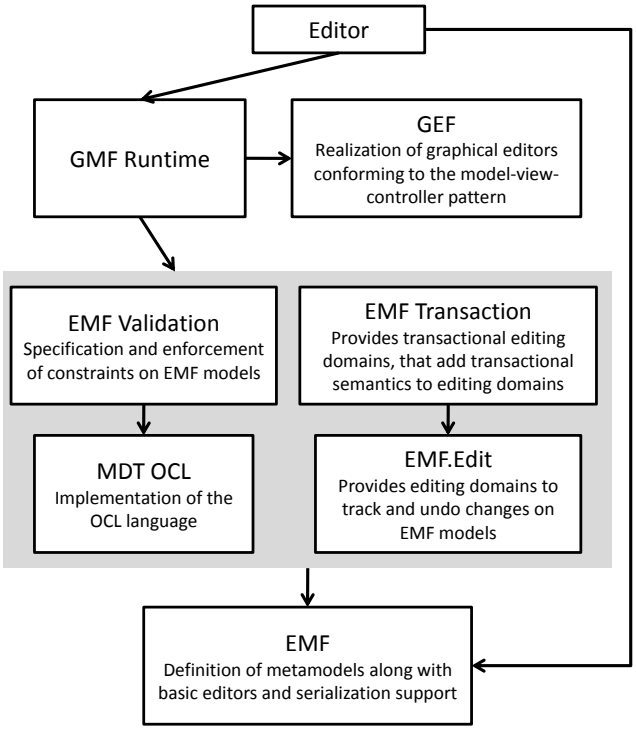

Figure A.4: Technologies employed by GMF Runtime.

clarity we omitted and continue to omit them. Let us note that in these drawings JET is never used: actually it is used during the generation of code for metamodels, not during the use of this code.

The reasoning done in this section also applies to figure 4.2 and 4.3.

B

DETAILED CHARACTERISTICS OF ANALYZED TOOLS

The details about the tools are to be found in table B.1, comprising the effort we spent on the study of each of them.

Tool	Version	Date	Hours spent on study
EMF	2.5	24/06/2009	47
GMF	2.2	24/06/2009	86
MDT UML2	3.0	24/06/2009	19
UML2Tools	0.9	24/06/2009	17
MDT Papyrus	r471 (svn)	20/08/2009	17

Table B.1: Table with analyzed tools and their relevant properties.

Related technologies are reported in table B.2. We recall that major number versions equal to 0 (zero) indicate that associated tools are in incubation phase and thus they are still developing processes and communities required by projects hosted in Eclipse.

Tool	Version	Date
EMF Transaction	1.3	24/06/2009
EMF Validation	1.3	24/06/2009
MDT OCL	1.3	24/06/2009
GEF	3.5	24/06/2009
Xpand	0.7	24/06/2009
JET	1.0	24/06/2009
QVTO	1.0	24/06/2009
EEF	cvs	09/07/2009

Table B.2: Table with related technologies and their relevant properties.

C

SOME ADVICES ON EMF AND GMF

Although it's not completely related to the aim of this thesis, we detail some tasks in EMF and GMF that challenged us for lack of documentation. This way we hope to help anyone who wants to approach these technologies.

C.1 EXTRINSIC ID

To render more robust references to elements of our models, EMF provides extrinsic ID, that is the automatic assignment of an UUID to each created element. The exact procedure to enable this for a particular metamodel is:

- in the genmodel click on the metamodel package. In the properties view change the "Model -> Resource Type" property to "XMI";
- generate the model code;
- under the X.util package locate the file `XResourceImpl.java` (in place of X substitute prefix and packages used for the generation). In the XResouceImpl class override the useUUIDs method so it returns `true`:

```
@Override
  protected boolean useUUIDs() {
    return true;
  }
```

- in `plugin.xml` add an extension to the org.eclipse.emf.ecore.extension_parser extension point and fill it with the same data of the extension to org.eclipse.emf.ecore.content_parser.

C.2 SHORTCUT MECHANISM

Enabling shortcuts in a GMF editor is not difficult, but some passages can be quite tricky. We first introduce the procedure for a simple scenario, in which in a editor E1 we want to use shortcuts to elements of the same metamodel X. In this scenario the steps are:

1) in the generator model, find the "Gen Diagram" element;
2) the property "Contains Shortcuts To" should be filled with the file extension specified for models conforming to X;
3) the property "Shortcut Provided For" should be filled with the model ID of the graphical editor. This may sound a little strange – we were expecting the file extension again. We investigated a little on this behaviour and we believe this is due to the fact that GMF editors work directly with notational elements (which imply determined semantic elements) and these are known only to graphical editors;

4) in the "Context Menu" element add an element named "Create Shortcut Action". This step is often overlooked in the documentation.

Suppose now we want in this editor E1 to reference elements from the metamodel Y and suppose also there is an editor E2 that works on metamodel Y. We should follow these steps:

1) in the generator model for E1, find the "Gen Diagram" element;
2) the property "Contains Shortcuts To" add the file extension specified for models conforming to Y;
3) in the "Context Menu" element ensure to have the element "Create Shortcut Action";
4) open the generator model for E2 and find the "Gen Diagram" element;
5) in this generator model the property "Shortcut Provided For" should be filled with the model ID of E1.

More information can be found in this wiki page: `http://wiki.eclipse.org/GMF_GenModel_Hints`

BIBLIOGRAPHY

[1] E. Miotto and T. Vardanega, "On the integration of domain-specific and scientific body of knowlege in Model Driven Engineering," in *Proceedings of Workshop on the Definition, Evaluation, and Exploitation of Modelling and Computing Standards for Real-Time Embedded Systems (STANDRTS'09)*, 2009.

[2] ——, "An assessment of candidate MDE technologies," 2010, submitted to *DATE2010 – 13th Conference and Exhibition on Design, Automation & Test in Europe*.

[3] "The assert-project — Final Report," ASSERT Project, Tech. Rep., December 2008. [Online]. Available: http://www.assert-project.net/IMG/zip/Assert-Project-Final-Report.zip

[4] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, February 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.58

[5] M. Bordin, M. Panunzio, and T. Vardanega, "Beyond ASSERT: Increasing the Effectiveness of Model-Driven Engineering," in *13th Eurospace Conference on Data Systems in Aerospace, Istanbul, Turkey*, May 2009.

[6] J. M. Nowacki, "Kurier font," Entry at CTAN, February 2007. [Online]. Available: http://tug.ctan.org/tex-archive/fonts/kurier/

[7] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, Sept.-Oct. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231146

[8] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering, 2007. FOSE '07*. IEEE Computer Society, 2007, pp. 37–54. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.14

[9] J.-M. Favre, "Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon," in *Language Engineering for Model-Driven Software Development*, ser. Dagstuhl Seminar Proceedings, J. Bezivin and R. Heckel, Eds., no. 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2005/21

[10] J. Miller, J. Mukerji, and Others, "MDA Guide Version 1.0.1," Object Management Group, Tech. Rep., 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[11] Object and Reference Model Subcommittee of the Architecture Board, "A Proposal for an MDA Foundation Model, ormsc/05-04-01," Object Management Group, Tech. Rep., April 2005. [Online]. Available: http://www.omg.org/docs/ormsc/05-04-01.pdf

[12] S. Kent, "Model Driven Engineering," in *Integrated Formal Methods – Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings*, ser. Lecture Notes in Computer Science, vol. 2335. Springer, 2002, pp. 286–298. [Online]. Available: http://dx.doi.org/10.1007/3-540-47884-1_16

[13] J.-M. Favre, "Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus," in *Language Engineering for Model-Driven Software Development*, ser. Dagstuhl Seminar Proceedings, J. Bezivin and R. Heckel, Eds., no. 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2005/13

[14] R. L. Glass, "Searching for the Holy Grail of Software Engineering," *Communications of the ACM*, vol. 45, no. 5, pp. 15–16, May 2002. [Online]. Available: http://doi.acm.org/10.1145/506218.506231

[15] I. Weisemoller and A. Schiirr, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," in *Models in Software Engineering: Workshops and Symposia at Models 2007 Nashville, Tn, USA, September 30-October 5, 2007, Reports and Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5002. Springer, 2008, pp. 47–58. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69073-3_6

[16] "Unified Modeling Language Infrastructure, version 2.2, formal/2009-02-04," Standard, Object Management Group, 2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/

[17] "Unified Modeling Language Superstructure, version 2.2, formal/2009-02-02," Standard, Object Management Group, 2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[18] "Papyrus," Software tool, July 2009. [Online]. Available: http://www.papyrusuml.org/

[19] "Common Facilities RFP-5: Meta-Object Facility, cf/96-05-02," Object Management Group, June 1996. [Online]. Available: http://www.omg.org/cgi-bin/doc?cf/96-05-02.pdf

[20] "Object Analysis & Design RFP-1, ad/96-05-01," Object Management Group, June 1996. [Online]. Available: http://www.omg.org/cgi-bin/doc?ad/96-05-01.pdf

[21] "UML Proposal to the Object Management Group, version 1.1, ad/97-08-02," September 1997. [Online]. Available: http://www.omg.org/cgi-bin/doc?ad/97-08-02.pdf

[22] "Meta Object Facility (MOF) Core Specification, Version 2.0, formal/06-01-01," Standard, Object Management Group, January 2006. [Online]. Available: http://www.omg.org/spec/MOF/2.0/PDF/

[23] "Request For Proposal: MOF 2.0 Core RFP, ad/2001-11-05," Object Management Group, November 2001. [Online]. Available: http://www.omg.org/cgi-bin/doc?ad/01-11-14.pdf

[24] "Request For Proposal: UML 2.0 Infrastructure, ad/2000-09-01," Object Management Group, September 2000. [Online]. Available: http://www.omg.org/cgi-bin/doc?ad/00-09-01.pdf

[25] K. Duddy, "UML2 Must Enable a Family of Languages," *Communications of the ACM*, vol. 45, no. 11, pp. 73–75, November 2002. [Online]. Available: http://dx.doi.org/10.1145/581571.581596

[26] "Eclipse Modeling Framework (EMF)," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/emf/

[27] Metacase, "MetaEdit+," Software tool. [Online]. Available: http://www.metacase.com

[28] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The Generic Modeling Environment," in *Workshop on Intelligent Signal Processing*. IEEE, 2001.

[29] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, "MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations," in *Model Driven Architecture – Foundations and Applications. Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4066, pp. 361–375. [Online]. Available: http://dx.doi.org/10.1007/11787044_27

[30] Jetbrains, "Meta Programming Systems," Software tool. [Online]. Available: http://www.jetbrains.com/mps/index.html

[31] Microsoft, "DSL Tools," Software tool. [Online]. Available: http://msdn.microsoft.com/en-us/vsx/cc677256.aspx

[32] ——, "Oslo," Software tool. [Online]. Available: http://msdn.microsoft.com/en-us/oslo/default.aspx

[33] "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 3, ptc/2009-05-13," Standard, Object Management Group, May 2009. [Online]. Available: http://www.omg.org/cgi-bin/doc?ptc/09-05-13.pdf

[34] "OMG Systems Modeling Language, Version 1.1, formal/2008-11-02," Standard, Object Management Group, November 2008. [Online]. Available: http://www.omg.org/spec/SysML/1.1/PDF/

[35] Artisan Software Tools, "Artisan Studio," Software tool. [Online]. Available: http://www.artisansoftwaretools.com

[36] "ISO/IEC WD4 42010, IEEE P42010/D6," Standard draft, ISO/IEC, IEEE, January 2009. [Online]. Available: http://www.iso-architecture.org/ieee-1471/docs/IEEE-P42010-D6.pdf

[37] M. W. Maier, D. Emery, and R. Hilliard, "Software Architecture: Introducing IEEE Standard 1471," *Computer*, vol. 34, no. 4, pp. 107–109, April 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=917550

[38] "Eclipse Modeling Project," Eclipse project, September 2009. [Online]. Available: http://www.eclipse.org/modeling/

[39] "Eclipse 3.5 "Galileo"," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/galileo/

[40] "Graphical Modeling Framework (GMF)," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/gmf/

[41] "Graphical Editing Framework (GEF)," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/gef/

[42] "EMF Transaction," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/emf/?project=transaction

[43] "EMF Validation," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/emf/?project=validation

[44] "MDT OCL," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=ocl

[45] "Object Constraint Language, Version 2.0, formal/06-05-01," Standard, Object Management Group, May 2006. [Online]. Available: http://www.omg.org/spec/OCL/2.0/PDF

[46] "JET," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=jet

[47] "Xpand," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=xpand

[48] "Operational QVT," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/m2m/

[49] "MDT UML2," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=uml2

[50] "UML2 Tools," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=uml2tools

[51] "MDT Papyrus," Software tool, July 2009. [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=papyrus

[52] "Topcased," Software tool, July 2009. [Online]. Available: http://www.topcased.org/

[53] "MOSKitt," Software tool, July 2009. [Online]. Available: http://www.moskitt.org/eng/

[54] "Papyrus Roadmap, version 0.1," February 2009. [Online]. Available: http://dev.eclipse.org/svnroot/modeling/org.eclipse.mdt.papyrus/trunk/doc/DevelopperDocuments/PapyrusRoadmapDescription_v2009-02-17.odt

[55] "Extended Editing Framework (EEF) Proposal," 2009. [Online]. Available: http://www.eclipse.org/proposals/eef/

[56] International Business Machines Corp., "Eclipse Platform Technical Overview," April 2006. [Online]. Available: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf

[57] "Ecore Tools," Software tool, September 2009. [Online]. Available: http://www.eclipse.org/modeling/emft/?project=ecoretools

[58] "EMFatic," Software tool, September 2009. [Online]. Available: http://www.eclipse.org/modeling/emft/?project=emfatic

[59] E. Merks, "Creating Children You Didn't Know Existed," Blog entry, January 2008. [Online]. Available: http://ed-merks.blogspot.com/2008/01/creating-children-you-didnt-know.html

[60] "SharedEditingDomain – How to share an editing domain between several GMF editors," October 2008. [Online]. Available: http://code.google.com/p/gmftools/wiki/SharedEditingDomain

[61] "Bug 174961 - No Actions/Menus created from gmftool-Description," Entry in Eclipse Bugzilla, February 2007. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=174961

[62] "Creating Robust Scalable DSL's with UML – Companion projects," Eclipse update site, August 2009. [Online]. Available: http://archive.eclipse.org/modeling/mdt/updates/EclipseCon2008-UML-Tutorial/site.xml

[63] J. Bruck and C. Damus, "Creating Robust Scalable DSL's with UML," *EclipseCon 2008*, 2008, the presentation can be viewed only with Internet Explorer. [Online]. Available: http://www.eclipse.org/modeling/mdt/uml2/docs/tutorials/EclipseCon2008_Tutorial_Creating_Robust_Scalable_DSL_with_UML_files/frame.html

[64] C. Mraidha and J. Bruck, "[news.eclipse.modeling.mdt.uml2] Static profile stereotype application problem," Newsgroup discussion, May 2009. [Online]. Available: http://www.eclipse.org/forums/index.php?t=msg&th=151966&start=0&S=4811449c8a6d5719dfe6d452f888dae5

[65] Christian W. Damus, "[news.eclipse.modeling.mdt.uml2] Re: EMF validation constraint target for Stereotype changes," Newsgroup article, September 2007. [Online]. Available: http://dev.eclipse.org/newslists/news.eclipse.modeling.mdt.uml2/msg01367.html

[66] Y. Tanguy, P. Tessier, and R. Schnekenburger, "Applied Stereotype implementation in Papyrus," CEA, Tech. Rep., July 2009. [Online]. Available: http://dev.eclipse.org/svnroot/modeling/org.eclipse.mdt.papyrus/trunk/plugins/developer/org.eclipse.papyrus.doc/cookbook/AppliedStereotypeImplInPapyrus_V1.2.doc

[67] B. Majewski, "A Shape Diagram Editor," December 2004. [Online]. Available: http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html

[68] "GEF Description2," Wiki article, April 2009. [Online]. Available: http://wiki.eclipse.org/GEF_Description2

[69] "GEF Programmer's Guide," Topic of Eclipse Help, August 2009. [Online]. Available: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html

[70] "Developer Guide to Diagram Runtime Framework," Topic of Eclipse Help, 2005. [Online]. Available: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Diagram%20Runtime.html

[71] "GMF Tutorial – Part 1," Wiki article, August 2008. [Online]. Available: http://wiki.eclipse.org/GMF_Tutorial

[72] A. Tikhomirov and A. Shatalin, "GMF Best Practices," *EclipseCon 2007*, March 2007. [Online]. Available: http://eclipsezilla.eclipsecon.org/show_bug.cgi?id=3739

[73] I. Vessey, "Problems versus solutions: the role of the application domain in software," in *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*. New York, NY, USA: ACM, 1997, pp. 233–240. [Online]. Available: http://doi.acm.org/10.1145/266399.266419

[74] A. A. Aaby, *Introduction to Programming Languages*, 1996. [Online]. Available: http://burks.brighton.ac.uk/burks/pcinfo/progdocs/plbook/index.htm

[75] "Eclipse Development Process," Eclipse Foundation, August 2008. [Online]. Available: http://www.eclipse.org/projects/dev_process/development_process.php

[76] J. Bézivin and O. Gerbé, "Towards a Precise Definition of the OMG/MDA Framework," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*. Los Alamitos, CA, USA: IEEE, November 2001, pp. 273–280. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ASE.2001.989813

[77] "Guide to Systems Engineeing Body of Knowledge," INCOSE and Lockheed Martin Corporation, Tech. Rep., 2004. [Online]. Available: http://g2sebok.incose.org/

[78] P. Leach, M. Mealling, and R. Salz, "RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace," Proposed Standard, July 2005. [Online]. Available: http://tools.ietf.org/html/rfc4122

application domain A collection of problems that have something in common, usually (but not always) the nature of the problem. (Definition cited from [73])

computation The application of a sequence of operations to a value to yield another value. A computation is usually performed by a computer. (Definition cited from Chapter 1 of [74])
Computation Independent Model (CIM) In MDA a model that represents the application domain in which a software system operates.
concern An area of interest in a system important to one or more stakeholders, for example reliability, performance and security. (Definition inspired by [36])

Domain Specific Language (DSL) A language that contains concepts related to a particular application domain.

Eclipse
Eclipse 1) An open source framework apt for the construction of development environment and applications.
Eclipse 2) A collection of open source projects that leverage the Eclipse framework.
Eclipse Modeling Framework (EMF) Framework contained in Eclipse which permits the definition of metamodels and the manipulation of conforming models.
edit part Class of GEF that is responsible for keeping the view consistent with the model and for translating user requests into proper model changes.
edit policy Class of GEF employed by edit parts to obtain the operations to do on the model to obey to a given user request.
extension point The mechanism used by a plugin in Eclipse to allow other plugins to extend its behaviour. An extension point is described through a schema, which indicates the data needed to extend the behaviour.

generator model In EMF and GMF terminology a model used to specify parameters of the transformation used to obtain the final code.
graphical definition model In GMF terminology the model which defines the elements supported by a graphical editor. In particular it specifies shapes, containment boxes, nodes and connectors.
Graphical Editing Framework (GEF) Framework contained in Eclipse, which permits the creation of sophisticated graphical editors according to the model-view-controller paradigm.
Graphical Modeling Framework (GMF) Framework contained in Eclipse which permits the creation of graphical editor for Eclipse Modeling Framework (EMF) models. It is composed of a Runtime part, which contains the building blocks, and of a Tooling part, which automates the construction of graphical editors through the use of models and transformations.

high-integrity system A system that should exhibit at least one of these properties: safety (the system does not harm humans and en-

vironment), security (information should be manipulated only by authorized entities), reliability (the system should function according to its specification). (Definition inspired by http://www.cs.york.ac.uk/hise/information.php)

incubation phase The phase of an Eclipse project in which process, community and technology are developed to bring the project to maturity. (Definition inspired by [75])

mapping model In GMF terminology the model which establishes, for each element in an Ecore model, its graphical appearance (taken from a graphical definition model) and its creation tool (taken from a tool definition model).
MARTE (Modeling and Analysis of Real-Time and Embedded systems) UML profile for the specification and the analysis of real-time systems.
Meta Object Facility (MOF) A language used to specify metamodels and endorsed by the OMG. Notably, UML metamodel is specified using and shares core concepts with MOF.
metametamodel A model that describes a language to specify metamodels.
metamodel A model that specifies a modeling language, in other words a model of a language for models. (Definition inspired by [9])
model A simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. (Definition cited from [76])
Model Driven Architecture (MDA) An MDE initiative endorsed by OMG that uses models in order to reach technological independence from software platforms.
Model Driven Engineering (MDE) Approach to software engineering that advocates the use of models and automated transformations in order to develop software systems with increased quality and productivity in the face of increasing richness of functionalities.
model-view-controller paradigm for the realization of graphical user interfaces that advocates the use of a model to store the data of interest, of a view to show them to the user and of a controller to intermediate changes and updates between them.

notational model In GMF terminology the model that describes how elements in the semantic model should be displayed to the user.

Object Constraint Language (OCL) A formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. (Definition cited from [45])

Platform Independent Model (PIM) In MDA a model that represents a software system abstracting away from technological choices and details.
Platform Specific Model (PSM) In MDA a model that represents a software system implemented on a particular software platform with determined parameters. A PSM is apt for static analysis.
plugin The smallest unit that makes up an application built using Eclipse.

programming language A notation to write programs, which in turns express computations. (Definition cited from Chapter 1 of [74])

real-time system A system composed of hardware (in particular sensors and actuators) and software that should monitor and control an external environment. The actions of a real-time systems are subject to temporal constraints.

scientific body of knowledge A set of sound and proved solutions to recurrent problems shared by different application domains. The quality of these solutions are assessed by a scientific community.
semantic model In GMF terminology the model that contains the data of interest (for example an UML model).
source code Artifact that specifies a program expressed in some programming language.
static analysis An automatic technique to infer approximate information about execution of a program looking only at its specification (let if be source code or a model).
system An interacting combination of elements viewed in relation to function. (Definition cited from [77])
systems engineering Interdisciplinary approach and means to enable the realization of successful systems. (Definition cited from [77])
Systems Modeling Language (SysML) UML profile designed for systems engineering.

tool definition model In GMF terminology the model which defines the tools used in a graphical editor to create and manipulate shapes.
transformation The action of obtaining one or more target models from one or more source models. Usually a target model is used to lower the level of abstraction of a source model.

UML profile Lightweight mechanism provided by UML in order to adapt the UML metamodel to specific needs. Simply put, UML profiles are packages containing stereotypes, which extend existing UML metaclasses with additional information.
Unified Modeling Language (UML) A language promoted by OMG to specify and describe software systems.
Universally Unique IDentifier (UUID) An identifier that is unique across both space and time, with respect to the space of all UUID. (Definition cited from [78])

view A work product (usually one or more models) that describes a system according to some concerns of interest. A view conforms to a viewpoint. (Definition inspired by [36])
viewpoint A work product (usually a series of models) that indicates the concerns, the languages and the models that a view should address and employ. (Definition inspired by [36])

ACRONYMS

ASSERT Automated proof-based System and Software Engineering for Real- Time applications

CEA LIST Commissariat à l'Énergie Atomique – Laboratoire d' Intégration des Systèmes et des Technologies
CHESS Composition with Guarantees for High-integrity Embedded Software Components ASsembly
CIM Computation Independent Model
CORBA Common Object Request Broker Architecture

DSL Domain Specific Language

EEF Extended Editing Framework
EMF Eclipse Modeling Framework

FP6 Sixth Framework Programme

GEF Graphical Editing Framework
GMF Graphical Modeling Framework

ID Identifier

JET Java Emitter Templates

MARTE Modeling and Analysis of Real-Time and Embedded systems
MDA Model Driven Architecture
MDE Model Driven Engineering
MDT Model Development Tools
MOF Meta Object Facility

OCL Object Constraint Language
OMG Object Management Group

PIM Platform Independent Model
PSM Platform Specific Model

QVTO Query/Views/Transformations Operational

RFP Request For Proposals

SVG Scalable Vector Graphics
SysML Systems Modeling Language

UML Unified Modeling Language
URL Uniform Resource Locator
UUID Universally Unique IDentifier

XMI XML Metadata Interchange
XML Extensible Markup Language