Maintainability of Transformations in Evolving MDE Ecosystems

Dissertation

presented to
the Department of Computer Languages and Systems of
the University of the Basque Country
in Partial Fulfillment of
the Requirements
for the Degree of

Doctor of Philosophy

("*international*" mention)

Jokin García Pérez

Supervisor: *Prof. Dr. Oscar Díaz García*

San Sebastián, Spain, 2014

This work was hosted by the *University of the Basque Country* (Faculty of Computer Sciences). The author enjoyed a pre-doctoral grant from the Government of the Basque Country under the "Researchers Training Program" during the years 2010 to 2013. The work was co-supported by the *Spanish Ministry of Science and Education*, and the *European Social Fund* under Contract TIN2011-23839 (Scriptongue).

A la abuela
María

Summary

Model-Driven Engineering (MDE) is a paradigm that uses models to develop software. These models conform to metamodels, and are transformed to other models or to code, building an ecosystem of related artifacts. In this context, maintainability becomes crucial to keep the different artifacts in sync. Evolution of an artifact should ripple along the dependent artifacts who are said to "co-evolve".

Within the MDE ecosystems, transformations play a preponderant role. This pivotal place makes them also specially prone to evolution. Model-to-model transformations are coupled to metamodels, and model-to-text transformations, to platform. This implies that upgrades in either of these two dependencies can make the transformation break apart. This is exacerbated by two main considerations. First, transformations tend to be complex programming artifacts. Unlike metamodels, transformation languages are far from being fully declarative, and still exhibit an algorithmic flavor. This makes transformation not only difficult to write but also to debug and maintain. Second, transformations tend to exhibit external dependencies, i.e. dependencies with artifacts which are outside the realm of the transformation programmer himself. In the case of model-to-model transformations, it is not odd for the metamodel team not to overlap with the transformation team. Skills are different, and this may lead to teams being split based on their familiarization with the domain (meta-modelers) *versus* the competence with transformation languages. Similarly for model-to-text transformations, platforms are often managed by third parties.

This Thesis addresses techniques and tools that help in maintaining transformations, specially focusing on keeping them in sync with the rest of the MDE ecosystem. Specifically, this Thesis' main contributions include:

1. a semi-automatic process to co-evolve model-to-model transformations upon metamodel evolution,
2. an adapter approach to make model-to-text transformations resilient upon platform evolution,
3. assisting in the testing of model-to-text transformations, measuring the completeness of the input model test suite, and debugging the detected errors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The software domain could be regarded as "huffing and puffing to catch up" with the increasing complexity of tasks which are subject to automation. More that twenty years ago (an eternity for this domain), [Bro87] introduces the key difference between *essential complexity*, i.e. this coming from the task itself, and *accidental complexity*, i.e. this originating from the instrumental stuff being used. One way of managing complexity that has been used in multiple domains, is abstraction [Sch06]. In software development, *Model-Driven Engineering (MDE)* [Sch06] is used to cope with accidental complexity, using models that capture the essential complexity that wants to be treated for a specific problem. MDE systematically uses models as the primary artifacts in the development process. Its popularity has increased in both academia and industry, as MDE claims many potential benefits (e.g. productivity, portability, maintainability and interoperability) [HRW11]. That said, MDE brings new challenges to both organization and software development. In particular, we focus on the way maintainability (i.e. adapting a product to a modified environment) is conceived in this paradigm. MDE rests on an ecosystem of artifacts where they are not isolated, but related to each

other [IPM12]. In this context, the maintainability of one artifact depends on the other artifacts. The challenge, hence, is to establish relationships between ecosystem artifacts and keep them coherently synchronized .

Several studies indicate that software maintenance accounts for at least 50% of the total production cost, and sometimes even exceeds 90% [Men08]. Software maintainability is needed for several reasons: to correct errors, to improve non-functional requirements, or to adapt it to a changing environment [IEE99]. Evolution is an inevitable phenomenon in software that needs to be taken into account from its inception, and MDE is not an exception. In MDE, maintainability is even *more* important, because in the context of software ecosystems, there are more sources of instability that can break the correction of the artifacts, as an artifact itself can change, but also its related artifacts. The MDE ecosystem includes metamodels, models, transformations, and the target technological platforms. Two of the most important instability sources come from the evolution of metamodels and platforms. Both severely impact transformations. This coupling is crucial, as transformations are considered as a core component in MDE, even the "heart and soul of MDE" [SK03]. Indeed, it is stated that MDE hardwires more architectural and design decisions than a traditional development platform, making multiple dimensions of evolution necessary [VWVDVD07]. However, little support exists for metamodel or platform evolution, jeopardizing the promise of MDE of reducing the costs of maintenance.

The following reasons sustain the importance of tackling maintainability in MDE:

- *Increase in the number of artifacts*. Traditionally, code when developed manually, is the only artifact that has to be maintained. Within the MDE ecosystem, system definition is split along different concerns (a.k.a viewpoints) and abstraction layers. This increases the number of dependencies to be kept in sync. Specifically, when the platform evolves, the code generators (i.e. transformations) and the application framework need to change to reflect the new target

platform [VWVDVD07]. In the case of metamodel evolution, both models and transformations are impacted.

- *Increase in the complexity of artifacts* . Not only is the number of impacted artifacts bigger, but also their complexity. A clear example are model-to-text transformations. These artifacts are more complex than the code they generate because their expressiveness includes the grammar of the code language, the grammar of the transformation language and references to the input model. This mixture and interleave of different types of concepts (e.g. java methods, control instructions from the transformation language, and concepts from the domain model whose value is not available at compile time) make the transformation difficult to read, and the mapping with the platform, less obvious.

- *Larger upfront investment*. The upfront investment in MDE is larger than the one required for developing a single application. MDE ecosystem (i.e. metamodels, models, and transformations) surpass that of traditional development. MDE presumes certain stability along time in order to reuse the infrastructure. Therefore, *Return Of Investment (ROI)* is obtained in the medium/long run as distinct generated applications benefit from the MDE infrastructure. Therefore, means are needed to make this infrastructure resilient against changes in "the exterior": the technological platform.

We abound as for the last point. Managing platform evolution in an MDE ecosystem becomes even more difficult when the dependency with the platform is external, i.e., the platform belongs to a different organization. When an ecosystem is open, the innovation is encouraged through open collaboration. But, on the other hand, it must be recalled that software components in the ecosystem might come from different partners, and that changes in one of the components will be usually out of the control of the rest of the partners, and might be accompanied by poor documentation, lost communication with the partner responsible for the change, and so

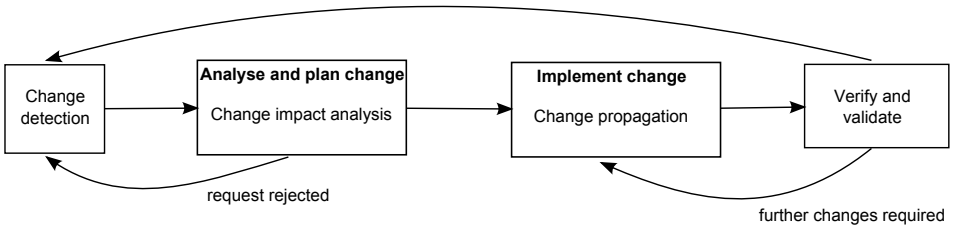

Figure 1.1: Evolution process (adapted from [Men08])

on. This problem exacerbates with the so-called “perpetual-beta systems”. Along Wikipedia, “perpetual beta is the keeping of software or a system at the beta development stage for an extended or indefinite period of time”. This is a defining characteristics of Web2.0 systems as stated by Tim O’Reilly: “The open source dictum, ’release early and release often’, in fact has morphed into an even more radical position, ’the perpetual beta’, in which the product is developed in the open, with new features slipstreamed in on a monthly, weekly, or even daily basis. It’s no accident that services such as Gmail, Google Maps, Flickr, del.icio.us, and the like may be expected to bear a ’Beta’ logo for years at a time." These are certainly not good news for MDE.

This calls for automate as much as possible all the necessary steps in MDE evolution, i.e. change detection, change impact analysis, change propagation and verification of changes done (steps defined in Figure 1.1). This Thesis focuses on offering some techniques and tools to developers in charge of maintaining the MDE infrastructure.

1.2 General Problem Statement

1.2.1 Context

Similarly to “traditional” software, MDE artifacts evolve over time as well. The evolution process followed by the software basically remains the same in MDE evolution (Figure 1.1): first, there exist a change that needs to be addressed, due to many reasons (i.e. error correction, more

functionality, etc.); next it is assessed what the impact in the system will be; then the change is propagated to the system in a coherent way; and finally, it is verified that the system works as expected. However, some MDE specificities make this road-map even more complex:

- Heterogeneous artifacts. There are models, metamodels, transformations, editors, the generated code, etc. They are quite different from each other, and it is likely that the person in charge of the modeling artifacts (i.e. transformations, editors, ...) will be different from the person working with the target platform and generated code. It could be even another person responsible for the domain at hand.

- Complex artifacts. The size and complexity of transformations make it difficult to maintain them. In the case of *model-to-model (M2M)* transformations, the semantic gap between the input and the output metamodel can be very big, requiring a complex logic which might be composed of many rules. In the case of model-to-text transformations, as commented above, there is an intrinsic difficulty due to the mixture of concepts that live together (concepts from the model, transformation language and specific platform).

- Internal dependencies. The division of the MDE infrastructure in layers according to the abstraction level has benefits, but also collateral effects: system information is spread around different artifacts (i.e. code, models, metamodels and transformations). There is an increase in the number of artifacts as well as their dependencies: models conform to metamodels, model-to-model transformations are written between metamodels, and model-to-text transformations generate code for a specific platform.

- External dependencies. Some artifacts (e.g. technological artifacts) might be outside the boundaries of the MDE. The decision about

evolving or not the platform is taken by a third party, alien to the organization managing the MDE infrastructure.

1.2.2 Problem

In the context of an evolving MDE ecosystem, transformations are one of the artifacts that suffer those changing forces, compounded by the fact that they can be the most complex artifacts of the MDE infrastructure. We outline transformation evolution along the road-map of Figure 1.1:

1. **Change detection phase**. There are many change sources. This Thesis focuses on two of them: metamodel and technological platform. The main difficulty is that the evolving artifacts may belong to different organizations. There might not be control over the evolution: only the initial and last state of the artifacts might be known.

2. **Analyze and plan change**. Consequences of the previously detected changes must be studied. Specifically,

   (a) metamodel evolution impacts model-to-model transformations.

   (b) platform evolution impacts model-to-text transformations. This kind of transformations has platform-specific code hardcoded in *print statements*, and references from the model are interleaved in these statements. When the target platform evolves, these statements becomes outdated.

3. **Implement Change.** Once the impact of changes has been established, those changes have to be propagated to restore the coherence between related artifacts. This propagation is cumbersome and error-prone, and in some occasions, frequent. This advices to assist as much as possible this process.

Figure 1.2: Relation between Adapter and HandyMOF

4. **Validate**. Here, we focus on model-to-text transformations due to their external dependency with technological platforms. We have found that performing this maintainability task is difficult due to the chasm between the *model-world* and the *text-world*. Here, three different kind of artifacts (i.e. model, transformation and code) need to be in sync. To check this synchronization, traceability is needed among artifacts. However, template-based model-to-text transformation languages do not provide a traceability between the transformation and the generated code. Typical questions or doubts that usually arise include: "Has this part of the transformation executed with this input model and did it generate the proper code?" or "What transformation line did generate this code fragment?". In addition, transformation coverage analysis (i.e. the extent to which the different model variants have been considered by the transformation) is also of interest to ensure a thorough validation.

1.2.3 Contributions

MDE ecosystems exhibit numerous dependencies among the involved artifacts. Hence, MDE maintainability involves preserving these dependencies upon artifact upgrades. This Thesis explores three different strategies to keep artifacts in sync, namely:

- Perfective approach. A perfective action is proposed to recover the *syntactical correctness* of the transformation, and to ensure its coherence with the rest of the system. A semi-automatic co-evolution process is proposed, which adapts the transformation to metamodel changes. This process needs some commitments: the

transformation has to be syntactically correct, i.e. it must compile correctly; transformation deletions must be the minimal necessary, in order to keep the expected behavior; and the process must be as automated as possible. Implementation-wise, the adaptation is achieved through a *Higher-Order Transformation (HOT)* [TJF+09].

- Preventive approach. To increase the lifetime of model-to-text transformations upon platform evolutions, a preventive mechanism is introduced. The idea is to use the *Adapter pattern* to adapt dynamically the *print* statements of the transformations to the new version of the platform. A platform-specific adapter is introduced, that given the differences between platform versions, checks for each statement if it has been impacted by any of the changes. If so, adapts it to the new platform. This approach is applied to database schema evolution. In order to dump the changes done to the generated code to the M2T transformation, it is done an impact analysis to show what kind of change has impacted what transformation location (line, column), saving it in a traceability model. HandyMOF tool visualize graphically this traceability. This relation between the adapter and HandyMOF tool can be seen in Figure 1.2.

- Supportive approach. Model-to-text transformations tend to be difficult to read and debug. More to the point, ascertaining the correspondence between transformation statements and the code they generate might not be easy. This correspondence is less obvious in the presence of conditional and iterative instructions as well as when the number of references to the model is high. This work introduces a mechanism for tracing code back to the transformation instructions that generates it. In this way, the developer will always know what part of the code has been generated by what part of the transformation, and vice versa. In addition, to facilitate a proper coverage of the transformation code, a black-box testing process in proposed.

The common thread of all the contributions, is to fill the lack of techniques and tools that developers have in the job of maintaining transformations, specially focusing on evolution.

1.3 Problem Statement for Transformation Co-evolution

1.3.1 Context

Metamodel evolution is a frequent situation as new insights about the domain are obtained. The kind of changes are similar to those of class diagram: class addition, class removal, class rename, extract superclass, etc [GJCB08].

1.3.2 Problem

One of the existing problems in model-to-model transformations that hinders their reuse is its coupling to metamodels. This coupling is caused, as we said, by the fact that transformations are defined at the metamodel level. This coupling makes them fragile to metamodel evolution. We can see this illustrated in Figure 1.3.

After a metamodel evolution, the transformation may not be syntactically correct anymore, impeding it to compile properly. There may be references to updated model elements or to elements that do not exist anymore; or new elements might need to be mapped. Therefore, syntactic correctness must be recovered. Depending on the size and complexity of the transformation, doing this task manually can be cumbersome and error-prone.

One fact that makes transformation co-evolution even more important than the widely studied model co-evolution to metamodel evolution [Wac07], is that transformations are programming intensive, and frequently more costly than its model counterparts [DRIP11].

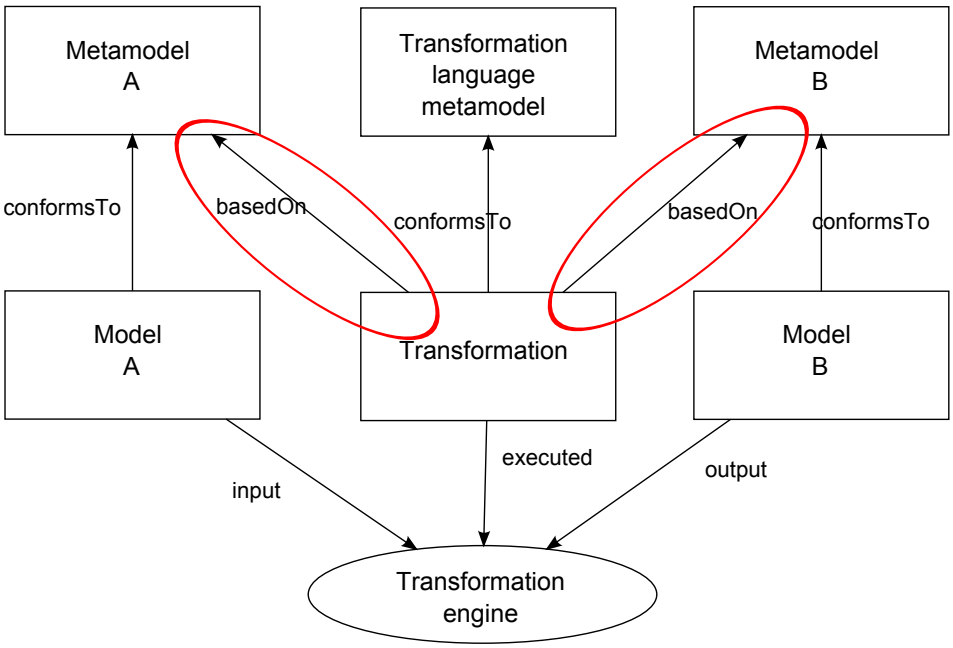

Figure 1.3: Coupling between M2M transformation and input and output metamodels

1.3.3 Contributions

A process is introduced to semi-automatically co-evolve transformations after metamodel evolution. This is borne out through a prototype for ATL [JABK08]. The process must guarantee that the references to the input or output metamodels are done to existing elements, i.e. the transformation compiles properly. Moreover, the adaptation process must not alter the expected behavior of the transformation. And last, whenever it is meaningful, new mappings can be inferred and added to the transformation in the case of an *additive* evolution (i.e. new elements are added). The process is divided into the following two main phases:

1. Detection phase. Simple changes between two versions of a metamodel (e.g. delete class, add attribute) are detected using a model comparison tool. Then, those simple changes are transformed into complex changes: i.e. changes that are semantically

meaningful.

2. Co-evolution phase. Co-evolving a transformation has different implications: (1) updated metamodel elements have to be updated in the transformation; (2) deleted metamodel elements have to be deleted as well in the transformation; and (3), new mappings may need to be added in the transformation in case of new elements in the metamodels. Fine-grained deletions are done in the transformation in a secure way, only deleting the necessary part. In the case of additive changes, metamodel matching is achieved between the source and target metamodels. Similarity is used to check if there exists an element equivalent to the new element added to the metamodel. In case a match is found, we will have the necessary information to generate the rule or corresponding binding. Otherwise, we opt to generate a rule skeleton with the available information. The process is semi-automatic as long as some changes can be handle automatically (*resolvable* changes) while other changes require human intervention (*unresolvable* changes) [CREP08].

1.4 Problem Statement for Adaptation of Generated Code

1.4.1 Context

Model-to-text transformations have platform-specific code hardcoded in *print statements*, and references from the model are interleaved in these statements. In this setting, if the target platform evolves, the transformation, and hence the generated code, becomes outdated. The notion of *"platform"* can be diverse and include code, database schemata, configuration files, documentation, etc. We are concerned with those that are likely to change, and whose evolution is out of our hands. For instance,

having this into account, an scenario that would be out of scope is the grammar of a language, because it does not happen frequently. Interesting scenarios are those of Web 2.0 platforms as wiki-engines, blog-engines, etc. which exhibit the perpetual-beta phenomena.

1.4.2 Problem

The impact of the evolution of technological platforms has been studied in the literature (for instance: API evolution [DJ06]), but not in a MDE context. One paradigmatic platform is a database. In the database world, the evolution of schemas has always been a concern. Databases clients, dependent upon the structures keeping the data, are impacted by the evolution [CH05]. Similar to the issue of keeping the application and database schema in sync, we focus on maintaining the consistency between code generators and the DB schema.

1.4.3 Contributions

An adaptability mechanism is proposed to shelter transformations from evolution in DB schema[1]. Next, we sketch the process:

1. During the detection phase, platform versions are turn into models (injection). The old version and the new version, now described as models, are compared which results in a difference model.

2. This difference model serves to feed the adapter (supported as a library) that permits to turn old statements into new statements at the time the code is generated, and without touching the transformation itself.

3. Apart from adapting the generated code, an impact analysis is done, creating a log and a model with the changes made and the traceability between the changed code and its position in the transformation.

[1] http://en.wikipedia.org/wiki/Adapter_pattern

This can later be accessed by the developer as an aid to upgrade the transformation code. This is a possibility available for the developer to decide whether she wants to transfer the changes made by the adapter to the transformation itself.

As a proof-of-concept, this approach is realized for the *Mediawiki* database as the platform, and the *WikiWhirl* transformation [PD12]. The "insert" statements of a SQL script that transforms a mindmap to a wiki are updated to the last version of the wiki engine. The adapter is platform-specific, and will only adapt SQL code, but it is domain-agnostic (it does not matter what type of DB schema has to be managed). Since the process is implemented in *Java* and *Ant*, a batch is available that allows to execute the whole process just with one single click: new database version installation, injection, comparison and adaptation. Preliminary validation has been conducted.

1.5 Problem Statement for Debugging and Coverage Analysis

1.5.1 Context

In addition to properly modeling the domain, MDE's most difficult task is developing a transformation. A transformation, as any other software product, must be designed, implemented and tested. Testing becomes even more important for transformations, considering that each transformation can potentially generate multiple applications in its lifetime. But testing is not only more important in the case of transformations, it is also more difficult, due to the intrinsic complexity of model-to-text transformations, as they combine grammars of the target platform with its own language structures, as well as the references from the input model.

1.5.2 Problem

Currently, when developing a model-to-text transformation, the developers do not have the appropriate tools to ensure their quality. Testing model transformations has proved to be a tough challenge [BGF+10]. Complexity of the transformations hinders its understanding, needed to trace back bugs to their causes (i.e. debugging). Developers only know what is the input and what have been the output of the transformation, but not the trace between them. In order to detect bugs and trace them back to their causes, a debugging mechanism to relate transformation and code is needed.

Another problem, is that testing the transformation for one input model does not guarantee that it will work with different model instances. To ensure that the transformation will work correctly in the future with different input models, it is needed to carry out a testing process with a coverage analysis that ensures that the model suite has covered all the transformation.

1.5.3 Contributions

Tracing mechanisms are proposed to ease the linkage between transformation instructions and generated code. These insights are realized for MOFScript. Specifically, an Eclipse plug-in is being developed (i.e. HandyMOF) that turns both transformations and generated code into hyper-documents. In this way, clicking on a transformation instruction highlights which code generates, and vice versa, i.e. clicking on a code instruction foregrounds the *print* instruction in the MOFScript transformation. In addition, the tool supports coverage analysis in assisting designers to come up with thorough test suits.

1.6 Outline

This section outlines the content of the Thesis. Figure 1.4 illustrates the chapters of the dissertation and the rest of the section gives an overview of

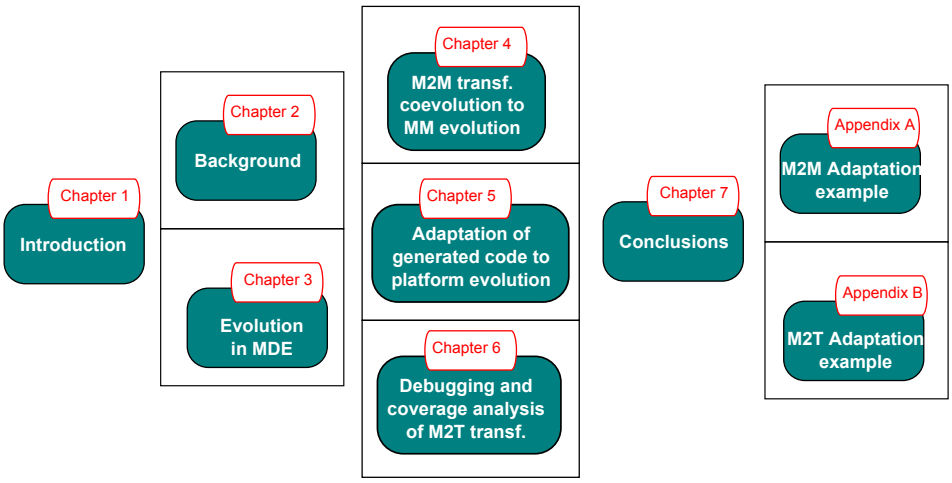

Figure 1.4: Chapter map

each of those chapters.

Chapter 2

This chapter introduces basic MDE ideas and explains concepts that will be used along the Thesis.

Chapter 3

This chapter explains in more depth MDE concepts related to evolution and maintainability that are more specific to this Thesis.

Chapter 4

This chapter presents an approach to adapt model-to-model transformations to metamodel evolution.

Chapter 5

This chapter presents an approach to adapt generated database code in a model-to-text transformation to platform evolution.

Chapter 6

This chapter presents a tool that assist the debugging and testing of model-to-text transformations

Chapter 7

This chapter concludes the Thesis by remarking the contributions and issues, listing the author's publications and suggesting possible future

work.

Chapter 2

Background

2.1 Overview-Introduction

This chapter will provide the reader the background needed to understand the grounds of the rest of the Thesis. As this dissertation faces maintainability and evolution issues in MDE, it will be first explained MDE in general, and then, specific maintainability and evolution concerns.

2.2 Model-Driven Engineering

2.2.1 Definition and Motivation

Model Driven Engineering is a software engineering approach that addresses platform complexity and the inability of third-generation languages to alleviate this complexity and express domain concepts effectively. For doing so, it combines:

1. *Domain-Specific Modeling Languages (DSML).* DSML are described using metamodels, which define concepts and their relationships within a domain; and

2. transformation engines that use models to synthesize software artifacts [Sch06].

In MDE, when abstract models are transformed to concrete implementations, it is called *Forward Engineering (FE)*.

MDE uses abstraction as a way to manage complexity of systems. Some elements from the conceptualization of the system are used to construct a model of it, depending on the requirements of the problem at hand. This model represents concepts (or entities) and relationships between them. MDE enacts models as first-class citizens in software engineering, following the "everything is a model" motto.

Software Engineering recommends to distinguish between the problem and solution spaces. The Problem Domain is an area or domain of expertise with specific concepts that is analyzed to resolve a problem. That Problem Domain is conceptualized in the Domain Model, which formalizes the concepts and vocabulary of the domain in a model. The Domain Model will describe the entities and their attributes and the relationships between them. On the other hand, the Solution Domain includes specific technologies that are used to resolve the problem at hand.

Modeling is not a software development approach that rejects programming. There is not a conflict between modeling and programming, and I would say that the distinction between both is artificial: models can be executable or interpreted as well. As Fowler says [FS03], the real question instead is to understand and define the right abstraction level for addressing each development activity.

Advantages and risks of MDE

MDE claims some advantages, but also suffers some drawbacks:

Advantages: Some of them from [MD08]:

- Separation of the specification of a system from the details how the system is implemented via concrete technologies. Many implementations using concrete technologies may be derived from the same abstract specification. Portability and reusability are achieved this way [Kur05].

- Reducing the abstraction gap between the problem domain and the software implementation domain. This is achieved through the use of transformations that support automatic transformation of problem-level abstractions to software implementations.

- Increase productivity and shorten development time. Maintenance and understanding of code is a difficult and error-prone process in case of large software systems. Automatic generation of code reduces time to market. Some studies have reported the productivity gain[1] [Mod03].

- Improve quality. Improve the quality of the generated code, improve the quality (assurance) of system requirements and manage requirement volatility, improve the quality of intermediate models, and earlier detection of bugs.

- Automation. A typical use of MDE is the automatic generation of code and other artifacts. But we must be aware that MDE is not only code generation: it includes also reverse engineering, interoperability, etc.

- Maintenance and evolution concerns. Maintain the architecture intact from analysis to implementation, evolution of legacy systems, concerns over software method and tool obsolescence, verification of the system by producing models from traces and that platform independent models have a considerable lifespan.

- Improved communication and information sharing. Between stakeholders and within the development team. Ease of learning.

- Standardization and formalism. Provide a common framework for software development across the company and phases of the life cycle that formalizes and organizes software engineering knowledge at a higher level of abstraction and a common data exchange format.

[1] http://www.omg.org/mda/products_success.html

- Better system understanding: Modeling harnesses creative solutions and helps to understand the system as a whole.

- Interoperability: Large systems are not monolithic but modular. Different modules are built upon technologies suitable for the problem at hand. Therefore, software systems consist of components implemented in various technologies that need to interoperate.

Disadvantages: Despite the advantages we have claimed, there is not silver bullet in software engineering. Some of the disadvantages are the following.

- Training cost: This paradigm shift involves technical changes (in the processes, tools, etc.), and its learning has a cost. This cost will be economical for the company at short-term, until the workers are productive with the new tools.

- Development cost: In my opinion, there is not a unique solution that fits all the problems: creating a MDE infrastructure has a cost, and there must be a close relationship with the company's organization to be aware that the infrastructure has to be used in several scenarios to make it profitable. It is recommended to conduct a Return of Investment (ROI) to estimate when will the project be profitable.

- Resilience of developers to change. There is a personal effort that developers will suffer too, as they must overcome the learning curve.

- Although empirical evidence has proven that MDE has benefits on maintainability, it is also agreed that there are negative influences, as the need to keep models/code in sync [HWRK11].

- Maturity of some tools.

- Over-modeling may be a problem, as it adds complexity [HWRK11].

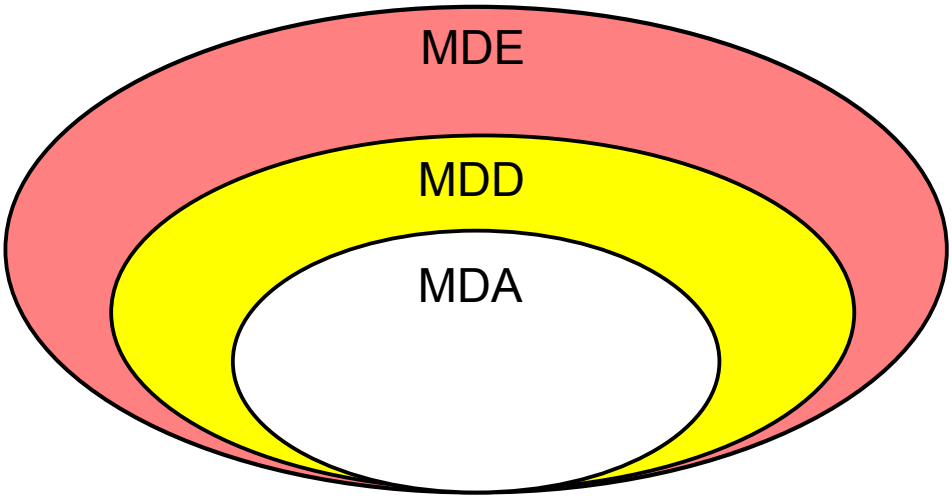

Figure 2.1: MD* acronym relationship

This Thesis is going to focus on maintenance and evolution concerns. As seen on the advantages and disadvantages, it seems to be a contradiction: on the one hand, MDE claims to improve the maintenance; but on the other hand, there is not empirical evidence of that. How is this possible? MDE is a relatively new paradigm, that lacks the diversity of tools needed for developers to take full advantage of the possibilities that MDE can bring. In the other hand, if there is not a big community of developers, companies are not interested in investing in MDE tools. This Thesis is trying to put its two cents in to go out of that vicious circle.

2.2.2 MD*: MDE, MDD, MDA

In Model-Driven Engineering, there is a jungle of acronyms that is convenient to clarify. All of them are based on the use of models in the development process, but they are differences in the degree of usage on the process. Relation of acronyms that are interesting for this Thesis, are expressed in Figure 2.1.

MDE

Model-Driven Engineering (MDE) would be an extension of MDD/MDA [Ken02] with the incorporation of a systematic process throughout the software life cycle, which is necessary in engineering. Processes introduce concepts, methods and tools, [Sch06] which in this case compromises models, metamodels and transformations.

MDD

Model-Driven Development (MDD) is a development paradigm where models are central in the development process. The main idea of MDD is that software is specified in models, that will define different views of the system. There will be also transformations that establish the mapping between models, and often, generate automatically the implementation artifacts.

MDA/ADM

Model-driven Architecture (MDA) is a framework for software development adopted by the Object Management Group (OMG) and thus relies on the use of OMG standards. Therefore, MDA can be regarded as a subset of MDD, where the modeling and transformation languages are standardized by OMG. Its metametamodel is known as *Meta Object Facility (MOF)*, which provides a language for defining the abstract syntax of modeling languages [BCW12].

OMG focuses Model-Driven Architecture on *Forward Engineering (FE)*. It advocates for code to be generated from abstract, human-elaborated modeling diagrams, dynamically through model-to-text transformations that target a specific platform.

MDA aims to separate the problem domain from the solution domain, in order to save the conceptual design from the changes in realization technologies [MM03]. These realization technologies are called *platform:*

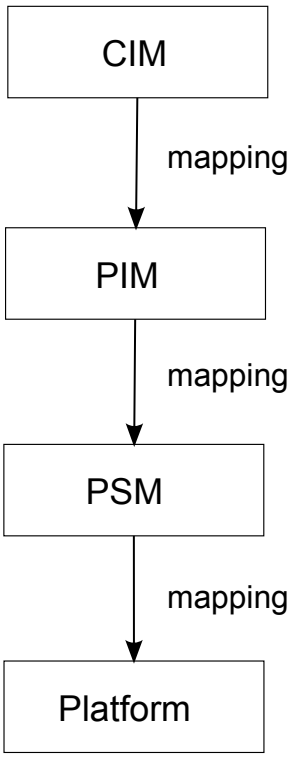

Figure 2.2: MDA abstraction levels and their relation

"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented"

MDA understands platform independence as a matter of degree. Different models might assume different abstraction levels from the platform. Related to this degree, they propose three viewpoints:

- Computation Independent Viewpoint: focuses on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden. A Computation Independent Model (CIM) is a view of a system from the Computation Independent Viewpoint.

- Platform Independent Viewpoint: focuses on the operation of a system while hiding the details necessary for a particular platform. A Platform Independent Model (PIM) is a view of the system from the Platform Independent Viewpoint.

- Platform Specific Viewpoint: combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system. A Platform Specific Model (PSM) is a view of a system from the Platform Specific Viewpoint.

On the other hand, Architecture-Driven Modernization (ADM) studies reverse engineering. ADM decodes to Architecture-Driven Modernization. The objective of ADM is to produce standards for model-based reverse engineering of legacy systems. The Abstract Syntax Tree Metamodeling (ASTM) and the Knowledge Discovery Metamodeling (KDM) are two complementary modeling specifications developed by the OMG Architecture Driven Modernization Task Force. ASTM provides high-fidelity low-level syntax models and their basic semantics, and the KDM provides the higher level semantic models of software [OMG11a].

- ASTM (Abstract Syntax Tree Metamodel): Establishes a specification for abstract syntax tree models. Thus, in contrast to other software representation standards, such as the KDM, the ASTM supports a direct 1-to-1 mapping of all code-level software language statements into low-level software models. AST is composed of a core specification, the Generic Abstract Syntax Meta-Model (GASTM), and a set of complementary specifications that extend the core, called the Specialized Abstract Syntax Meta-Models (SASTMs). The ASTM is expected to complement the KDM.

- KDM (Knowledge Discovery Metamodel): it is a meta-model with a very broad scope that covers a large and diverse set of applications, platforms, and programming languages [OMG11b].

In [DGB12] they state that the link between ASTM and KDM is often fuzzy, or even unestablished since KDM is in charge of synthesizing all captured software artifacts. They introduce a bridge to reconcile both standard.

2.2.3 Artifacts

Next, the main elements of MDE are explained.

Models

There are many definitions of models both in general literature and in computer science in particular. The word *model* comes from the Latin word *modulus*, which means "small measure". And in any present dictionary can be found a definition like the following[2]: a representation, usually on a smaller scale, of a device, structure, etc. Generally speaking, a model is an abstraction of a conceptualization of a reality.

Models play an important role in sciences, architecture or engineering. They have been used from antiquity to understand better a complex problem and assess potential solutions before investing the resources needed to carry out a complete implementation. In each of the communities, there are variations in the understanding of what a model is. Even inside the software engineering community, different definitions have been given [MFB09]. The OMG defines a model as [MM03]:

"A model of a system is a description or specification of that system and its environment for some certain purpose. It is presented in a (modeling) language"

According to [Sel03], models must fulfill the following five characteristics:

- *Abstraction*. A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for

[2]http://www.collinsdictionary.com/

a given viewpoint, it permits us understand the essence more easily.

- *Understandability*. A good model provides a shortcut by reducing the amount of intellectual effort required for understanding.

- *Accuracy*. A model must provide a true-to-life representation of the modeled system's features of interest.

- *Predictiveness*. We should be able to use a model to correctly predict the modeled system's interesting but non obvious properties, either through experimentation or through some type of formal analysis.

- *Inexpensiveness*. A model must be significantly cheaper to construct and analyze than the modeled system.

Models are the main artifacts of MDE.

Metamodels

As we have said, models abstract elements from reality. In the same way, they idea of using models to abstract something can be applied also to models. In this sense, a metamodel is an abstraction of a model. OMG defines a metamodel [OMG02] as:

"A metamodel is an "abstract language" for describing different kinds of data; that is, a language without a concrete syntax or notation."

It defines the modeling language, as they delimit the models that can be represented. This language definition is abstract, in the sense that there is not a concrete syntax or notation. Metamodels define constraints that every model written in that language must satisfy. Therefore, they delimit the expressivity of the model to a specific domain.

Metamodels facilitate separation of concerns. When dealing with a given system, one may work with different views of the same system, each characterized by a given metamodel [BÓ4], which can later be composed into an integrated application [KR03].

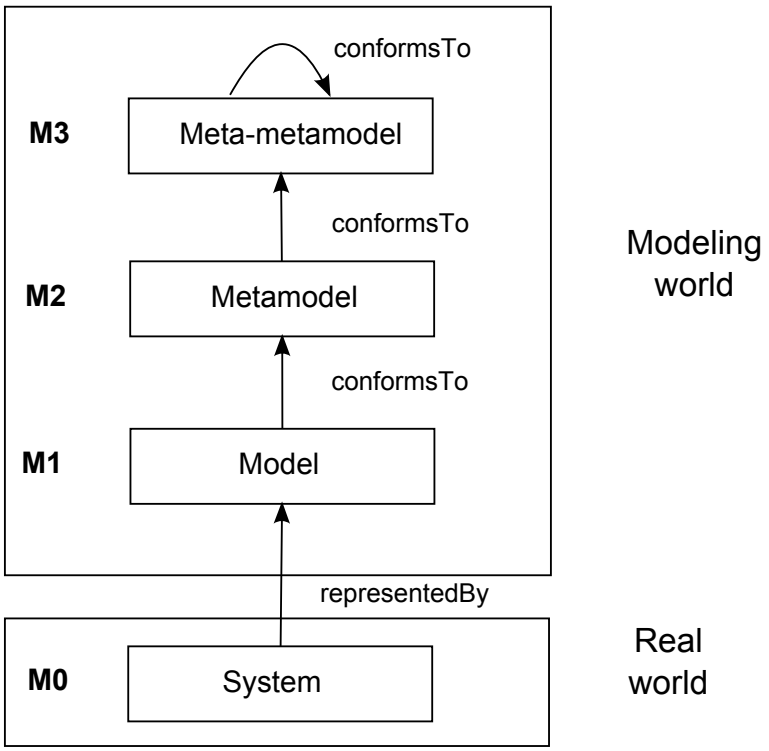

Figure 2.3: Four layer architecture: relation between reality, model, metamodel and meta-metamodel

In the same way that there are models that describe models, there are models that describe metamodels, which are called meta-metamodels. This meta-metamodels are reflexive (they are defined based on themselves), so there is not need for another abstraction level. The relation between a model and its metamodel is of *conformance*, and the relation of the metamodel with one of its models is of *instantiation*. These relations are analogous as the *inheritance* and *instantiation* from Object Oriented world.

The OMG defined a four layer architecture for metamodeling [OMG02], which is presented in Figure 2.3. M0 level represents the real system. Next level (M1) represents a model view of such system. In layer M2 there is the metamodel that describes the model in layer M1. And finally, M3 is the meta-metamodel which metamodel conforms to. This meta-metamodel conforms to itself.

EMF allows the definition of metamodels based on the metamodeling language called Ecore [BCW12]. EMF also provides generator components for producing from metamodels (i) a specific Java-based API for manipulating models programmatically and (ii) modeling editors to build models in tree-based editors. EMF comes with a powerful API covering different aspects such as serializing and deserializing models to/from XMI as well as powerful reflection techniques.

Model Transformations

In addition to models, the other basic artifact in MDE are transformations. The definition of model transformation given in the MDA guide is the following:

"Model transformation is the process of converting one model to another model in the same system"

The specification of this process is called *mapping* in the MDA guide. In this thesis, it is extended this definition to include also text as a target in the transformation. Therefore, for us, transformations establish a mapping between a model that conforms a metamodel, and another model or text. When the mapping is between two models, they are called *Model-to-Model (M2M)* transformations. When the mapping is between a model and text, the transformation is called *Model-to-Text (M2T)* transformation. Transformations are not glue-code, but reusable first-class artifacts, that can be enacted for a variety of input models that yield different output models.

Transformations can be used in several scenarios, with different goals. Some of them are the following [CH06]:

- Generating lower-level models, and eventually code, from higher-level models.
- Mapping and synchronizing among models at the same level or different levels of abstraction.

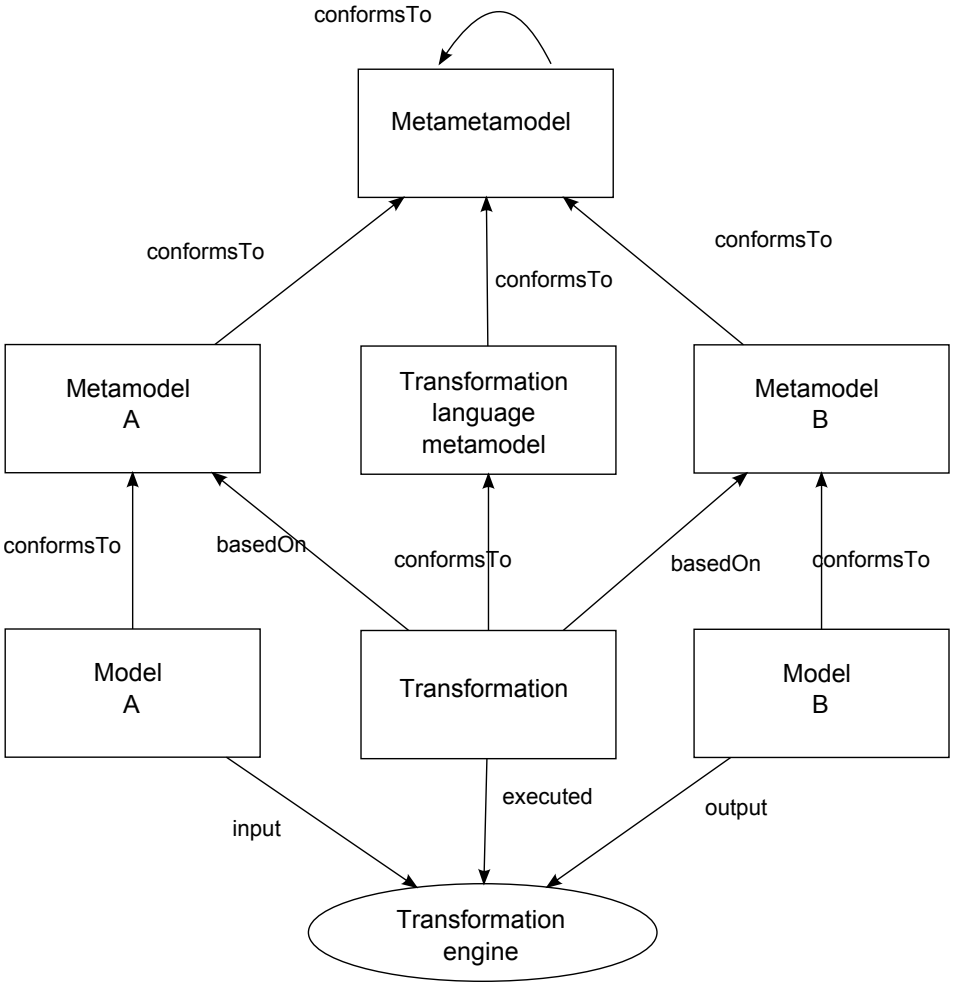

Figure 2.4: M2M transformations

- Creating query-based views of a system.
- Model evolution tasks such as model refactoring.
- Reverse engineering of higher-level models from lower-level models or code.

Then, Model-to-Model and Model-to-Text transformations are defined:

Model-to-Model Transformations

M2M transformation is a program which takes one or more models as input to produce one or more models as output. The number of input

and output models is variable: they might be *one-to-one* transformations, having one input model and one output model; *one-to-many*, *many-to-one*, or even *many-to-many* transformations. M2M transformations are executed between models, but are defined at metamodel level (observe the *basedOn* relation in Figure 2.4).

When both input and output models conform to the same metamodel, it is an *endogenous transformation*; and when they conform to different metamodels, it is an *exogenous transformation*.

There are different types of transformation languages to implement M2M transformations: declarative, imperative or hybrid.

- Declarative: They specify relationships between the elements in the source and target models, without dealing with execution order. Relationships may be specified in terms of functions or inference rules. In my opinion, this kind of languages offers a strong formal basis, which allows the validation of the transformation and the synchronization between models, but are limited and difficult to use when there is a big semantic gap between input and output metamodels. Some of the available tools are:

  - QVT Relational[3], that is a bi-directional declarative language proposed by the OMG.
  - Triple Graph Grammars (TGG), which are based on graph transformations. There are several implementations of this approach [And94].
  - Janus Transformation Language (JTL)[4], a bi-direccional transformation language.
  - EMFTiger[5]: Eclipse project that supports graph transformations of EMF models.

[3]http://wiki.eclipse.org/M2M/QVT_Declarative_(QVTd)
[4]http://jtl.di.univaq.it/
[5]http://user.cs.tu-berlin.de/~emftrans/

  - Henshin[6]: Successor of EMFTiger that introduces some advanced features such model checking support.
  - Fujaba[7]: It implements TGG for incremental model transformation and synchronization.
  - e-Motions[8]: It is an Eclipse plugin to graphically specify the behavior of modeling languages by using graph transformation rules

- Imperative: They specify an explicit sequence of steps to be executed in order to produce the result. QVT Operational[9] is a uni-directional imperative language proposed by the OMG.

- Hybrid: They combine declarative and imperative constructs. Available tools are:

  - ATL[10] has become the *de facto* standard for M2M transformations. It is a rule-based language heavily based on OCL [JABK08].
  - RubyTL[11]: it is a transformation language embedded in Ruby, that allows to use the elements of the host language [CMT06].
  - Epsilon[12]: It implements a task-specific rule definition and execution scheme, but also inherits imperative features of EOL (Epsilon Object Language) to handle complex transformations when necessary.

Some techniques have been also proposed to ease the development of transformations [LWK10, WKK+10].

[6] http://www.eclipse.org/modeling/emft/henshin/
[7] http://www.fujaba.de
[8] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions
[9] http://wiki.eclipse.org/M2M/Operational_QVT_Language_(QVTO)
[10] https://www.eclipse.org/atl/
[11] http://rubytl.rubyforge.org/
[12] http://www.eclipse.org/epsilon

Model-to-Text Transformations

Model-to-Text transformations are transformations that have one or several models as input, and text as output. The main application of this kind of transformations is to generate fully or partially the artifacts for a specific platform. These artifacts can be code, documentation, configuration files, etc. This is the last step in Forward Engineering processes, and involves a change from the "model world" to the "text world". Model-to-Text transformation languages are composed of three kind of instructions: 1) static parts that prints the text in the output the same way it has been written; 2) references to the model, that will retrieve the value from the input model; and 3) control instructions that control the execution flow (conditional expressions and iterative expressions). These are some transformation language tools:

- MOFScript[13]: This transformation language has a metamodel and an injector and extractor that allow to inject the transformation to a model conforming the transformation language metamodel, and to extract the model back again to text format. It also generates traceability between elements of the input elements and code.
- Acceleo[14]: It implements the MOF Model to Text Language (MTL) standard of the OMG.
- Xpand[15]: It is a statically-typed template language, which provides a mixture of Java and OCL [Kla08].
- JET[16]: Template-based language with syntax similar to JSP.
- Epsilon Generation Language (EGL)[17]: It is a template-based language that allows to make Java calls, which can be very useful in some scenarios [RPKP08].

[13]http://modelbased.net/mofscript/
[14]http://www.eclipse.org/acceleo
[15]http://www.eclipse.org/modeling/m2t/?project=xpand#xpand
[16]http://www.eclipse.org/modeling/m2t/?project=jet#jet
[17]http://www.eclipse.org/epsilon/doc/egl/

Domain-Specific vs General Purpose Modeling

When modeling a system, there are two main trends:

- *General-Purpose Languages (GPL)* or *General-Purpose Modeling Languages (GPML)* can be used in any domain to resolve any kind of problem. An example of GPL is Java, and in the case of GPML, *Unified Modeling Language (UML)*. UML is defined by the Object Management Group (OMG) [OMG]. UML have been applied to many domains and many implementation platforms. One of its advantages is that there are many mature tools available. However, such a general language may not be suitable for some scenarios: here is where the Domain-Specific Languages come.

- *Domain-Specific Languages (DSL)* or *Domain-Specific Modeling Languages (DSML)* when it is a modeling language, refers to languages that are used in a concrete domain, to achieve focused tasks in that domain. Therefore, the notation is very close to the problem domain. Well-known examples of DSLs are Matlab or SQL. To develop a DSL two alternatives exist:

  - Specialize UML with *UML Profiles*, extending UML original semantics for particular application domains [FV04].
  - Define a completely new language. DSLs can be created with tools like: XText[18] or EMFText[19].

Note that the goals of the two kind of languages are different. Therefore, both are complementary approaches and each of them should be used when appropriate.

[18] http://www.eclipse.org/Xtext/
[19] http://www.emftext.org/

2.2.4 Operations

In the MDE process several operations can be applied to models, including: merging, refactorization, verification, etc. Most of them are specific types of transformations. Then, operations used in this work are explained:

Comparison

When comparing two text files, text difference tools are used to retrieve changes between two files in terms of changed lines or characters. In the case of models, more useful that using this type of comparison, is to compare models as the graphs they are. Not only textually, but structurally. Graph comparison is a NP-problem, and to guarantee that it is done in a reasonable amount of time, comparison tools use heuristics. Model comparison is crucial for evolution management, to retrieve the differences between the new and the old versions of a model. Comparison tools take two models as input and output a difference model.

There are two conceptually different types of approaches for the representation and calculation of model differences:

- In the state-based approaches, the model differences are calculated between two states of a model, i.e. between two versions of a model. They use a difference model to describe those differences.
- In the operation-based (also called change-based) approaches, the model differences are represented by a sequence of predefined operations, which when applied to the initial model, produce the final model. Thus, in the operation-based approaches, all the tools used to develop models must supply a set of reusable coupled operations in a predefined form, while in the state-based approaches this is not necessary [vdBPV11]. This operations work both at metamodel level as well as at the model level [HVW11].

The advantage of state-based over the operation-based is that sometimes you are not the person/organization that evolves the artifact (it is an

external dependency in your system), and therefore, you do not have control on the change process to decide which type of changes are going to use or to record the evolution.

Some examples of tools are the following:

- EMF Compare[20]: Provides comparison and merge facility for any kind of EMF model. It includes a generic comparison engine [Tou06].
- AML: it allows to express matching strategies which compute mappings between models by executing a set of heuristics [GJCB09a].
- SiDiff[21]: It is a metamodel independent approach for model comparison.
- Epsilon Comparison Language (ECL)[22]: It is a DSL that enables users to specify comparison algorithms in a rule-based manner to identify pairs of matching elements between two models.

Based on my own experience, whenever it is possible, the use of an Universally Unique Identifier (UUID) in models is advisable to avoid some ambiguities in the comparison. Typical ambiguity case is confusing an update with an deletion + addition in some scenarios. This way, the comparison is simplified, as the comparison engine can match elements with the same ID.

Injection

Injection is a Text-to-Model (T2M) transformation where the input is a textual artifact and the output is a model. There are several tools that eases this process:

[20] http://www.eclipse.org/emf/compare/
[21] http://pi.informatik.uni-siegen.de/sidiff/
[22] http://www.eclipse.org/epsilon/doc/ecl/

- MoDisco (Model Discovery): is an extensible framework for model-driven reverse engineering, supported as an Eclipse Generative Modeling Technology (GMT) component [BBJ+10]. Its objective is to facilitate the development of tools ("discoverers" in MoDisco terminology) to obtain models from legacy systems during modernization efforts. XML and Java discoverers are available.

- Gra2Mol (Grammar-to-Model Transformation Language): is a Domain-Specific Language tailored to the extraction of models from GPL code. This DSL is actually a text-to-model transformation language which can be applied to any code conforming to a grammar. It aims to reduce the effort needed to implement grammarware-MDD bridges [CIGM14].

- API2MoL: Injects Java APIs to models, allowing the homogeneous treatment of all APIs [CJCGM11].

- Jamopp: Injects/extracts Java code to/from a Java model conforming a Java metamodel [HJSW09].

HOT

A *Higher-Order Transformation (HOT)* is a model transformation such that its input and/or output models are themselves transformation models. Following the "everything is a model" MDE motto [Béz05], transformations can be seen as models too [BBG+06]. This model representation of the transformation, allows to be modified by another transformation just like any ordinary model. This unification capability of models allows reusing tools and methods, and reusing the same framework. In order to inject the transformation to a model, the transformation language must have a transformation metamodel and an injector. Unfortunately, not all the transformation frameworks fulfills this requirement. In the case of ATL, there exist both the metamodel and

the injector. As ATL has been used along the work, I could have taken advantage of this capabilities.

There have been documented different use scenarios for HOTs [TJF+09]. As a sample we can cite:

- Transformation analysis: The HOT takes as input a transformation, and outputs some measures about the transformation.
- Transformation composition: The HOT composed the input transformations into the output transformation.
- Transformation modification: This HOT has an input transformation, that alters into an output transformation. There are different types of "Transformation modification" that can be classified:
  - Aspect weaving: The HOT weaves cross-cutting concerns into a model transformation.
  - Transformation refactoring: The HOT would encapsulate the refactoring, that would be executed by the user during the development of the transformation.
  - Transformation optimization: The HOT would analyze the transformation, and translate into an equivalent but more efficient transformation.

2.3 Maintainability and adaptability

2.3.1 Maintainability

In the Swebok[23] a number of software maintenance categories are described. Also in The Standard for Software Maintenance [Men08], which is part of the IEEE 1219, there is a definition of software maintenance:

[23] http://www.swebok.org

> *"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."*

The ISO/IEC standard for software maintenance [Men08] proposes four categories of maintenance:

- *Perfective maintenance* is any modification of a software product after delivery to improve performance or maintainability.
- *Corrective maintenance* is the reactive modification of a software product performed after delivery to correct discovered faults.
- *Adaptive maintenance* is the modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.
- *Preventive maintenance* refers to software modifications performed for the purpose of preventing problems before they occur.

2.3.2 Adaptability

The ability of a software system to cope with changes is an important characteristic that determines the adaptability of the system. *Adaptability* is a quality property of software products. Software product quality is treated in ISO 9126 [ISO01] which is an international standard. It defines six quality characteristics that are further refined into sub-characteristics. Adaptability is defined as a sub-characteristic of portability. The definition of adaptability according to ISO 9126 is the following:

> *"The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered."*

2.4 Testing in MDE

Taking into account the difficulty of providing formally correctness of software using formal verification techniques, an alternative approach widely applied in the industry is validation by testing [KAER06]. The aim of testing is finding errors in a program. The methodology to test a piece of software generally comprises a number of well-known steps: creation of input test cases, running the software with the test cases, and finally, using an oracle to analyze the results yielded to determine whether errors came up or not. An oracle is any program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object and it can be as simple as a manual inspection or as complex as a separate piece of software [GC12]. Three main approaches are known in testing:

- Black-box testing[24]: Examines the functionality of an application without peering into its internal structures. Only the input and the output of the program are considered.
- White-box testing[25]: Tests internal structures of an application. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs.
- Grey-box testing[26]: Is a combination of white-box testing and black-box testing. The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications.

In MDE, testing is used as well to test transformations. However, the nature of the input and output data manipulated by transformations makes these activities more complex. Indeed, transformations manipulate models, which are very complex data structure. This makes the problem of test data generation very difficult in the case of model transformations [BDTM+06]. Validation of model transformations is important for

[24]http://en.wikipedia.org/wiki/Black-box_testing
[25]http://en.wikipedia.org/wiki/White-box_testing
[26]http://en.wikipedia.org/wiki/Gray_box_testing

ensuring their quality. When coding a conceptual transformation rule, the following errors can occur [KAER06]:

1. Meta model coverage is insufficient.
2. Creation of syntactically incorrect models: they do not conform to the metamodel or violate constraints.
3. Creation of semantically incorrect models.
4. Confluence: if two rules are not parallel independent, they might give rise to confluence errors.
5. Correctness of transformation semantics: the transformation does not preserve a desired property.
6. Errors due to incorrect coding: classical coding errors.

2.4.1 Black-box approaches

A metamodel captures the set of all the possible valid models for a modeling language. We call this set the *modeling space* [SBM08, CBS12]. The aim is to build automatically or manually a set of input models, as small as possible, but at the same time being a good representative of the whole input space [TRL12]. In black-box testing approach a set of test models that conform to the constraints defined by the metamodel is generated. Black-box is independent of any specific model transformation language. This generation can be done manually or automatically.

In manual approaches, they assume that tests are planned by the user and defined in a textual test specification [LZG05]. Sometimes the designer has to specify pre/post conditions and invariants of the transformation to generate the input models [Gue12]: this can be considered a semi-automatic approach.

On the other hand, in automatic approaches, they present algorithms that given a metamodel as input, they generate a set of test models

[BFS+06]. Test models should instantiate each class and each relation of the input metamodel at least once. In order to determine relevant values for the properties of objects (attributes and multiplicities), it is adapted a classical testing technique called *category-partition* testing. The idea is to decompose an input domain into a finite number of sub-domains and to choose a test datum from each of these sub-domains. For example, there are boolean partitions where {{true}, {false}} designates a partition with two ranges: true and false. Each range of each partition for all properties of the meta-model must be used in at least one model fragment [FBMT09]. There are tools that generate automatically those partitions. For instance, in [SBM08] they developed a tool called MMCC (Meta-model Coverage Checker) that can generate model fragments, according to a particular criterion, from any metamodel. One of the techniques that has been proposed for generating meaningful partitions is knowledge-based partitioning: it consists of extracting representative values from the model transformation [FSB04].

The input metamodel for a transformation is usually larger than the actual metamodel used by the transformation: the part of the metamodel that has been used is called *effective metamodel*. In order to know the effective metamodel, the idea is to collect the meta-model elements that are referred or used in the transformation.

Determine the quality of the test-suite: coverage-analysis

A model test suite is said to achieve metamodel coverage, if all of its model elements (e.g. class, feature, inheritance and association) are covered [WKC06] (and not only the effective metamodel). Metamodel coverage helps the user assess which parts of a metamodel are referenced by a given model transformation set.

To determine the quality of a test suite, coverage analysis can be used. There are tools [KKBG13] that can be used for measuring coverage achieved by test model transformations.

2.4.2 White-box approaches

As white-box testing is specific to a concrete language, there have been proposed solutions to specific languages. In the process, data flow graph is generated in the first place. The second step is to traverse the dependency graph a number of times. Traversing the dependency graph implies setting truth values for the different conditions in the graph and, therefore, each traversal will yield a set of constraints that symbolizes a family of relevant test cases for the transformation. In the last step, the actual test cases (i.e. the test input models to be used when executing the transformation) are created by computing models conforming to the source metamodel and satisfying the constraints for the test case [GC12].

Regarding the analysis of internal structures needed in white-box testing, it can be used static analysis to guide the automatic generation of test inputs for transformations [MSTC12]. That static analysis uncovers knowledge about how the input model elements are accessed by transformation operations. This information is called the input *metamodel footprint*. Footprint is transformed, with input metamodel, its invariants, and transformation pre-conditions to a constraint satisfaction problem.

Chapter 3

Evolution in MDE

3.1 Introduction

Software evolution is an inevitable process where software systems need to be continually adapted to the changing environment. MDE is not an exception. MDE artifacts rarely exist in isolation. In fact, they are often tightly coupled: models conform to metamodels, transformations are defined from a metamodel to another metamodel (or code), code is generated for a particular platform, etc. In this sense, we can say that modeling artifacts live in an ecosystem [IPM12]. Architecturally, a software ecosystem consists of a platform, products built on top of that platform, and applications built on top of the platform that extend the products with functionality developed by external developers [Bos09]. While software ecosystems offer numerous advantages [JAOA13], their development also creates a network of complex interdependencies between elements throughout the whole ecosystem. If we focus on MDE ecosystem, models, metamodels, and transformations are heavily interrelated by foundation.

Software ecosystems evolve, and co-evolution denotes the necessary evolutionary mutual changes of software components that interact with each other after one evolves.

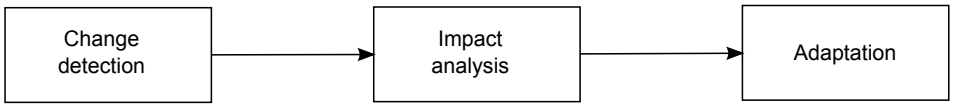

Figure 3.1: Adaptation process

While introduction of model-driven engineering brings advantages, it also requires a new style of evolution. In traditional software evolution, the development platform is fixed. Since an MDE platform hardwires many more architectural and design decisions than a traditional development platform, platform evolution is a requirement for MDE [VWVDVD07]. Thus, MDE requires multiple dimensions of evolution, including metamodel evolution and platform evolution, that will be explained in the next subsections.

Moreover, customization on generated code brings more maintainability problems. But in this case, instead of considering it a problem of MDE, we put the responsibility on developers, that must follow the best practices. They should customize and enrich models instead of modifying generated code.

3.2 Co-evolution, Synchronization and Adaptation

MDE is an ecosystem, where the artifacts are related to each other: models conform to metamodels, M2M transformations are defined between metamodels, M2T transformations generate code for a specific platform and model editors conform to a metamodel. When one of the artifacts evolves, the system may have become inconsistent. Related artifacts must co-evolve accordingly to recover the *consistency*. In order for the co-evolution to be meaningful, moreover to the consistency, the system must be semantically equal to its previous version.

Normally, roughly, the typical adaptation processes follow the schema

that is presented in Figure 3.1:

1. Change detection: there is a first phase where the difference between evolving artifact versions are retrieved.

2. Impact analysis: once that relationships between different versions have been made, it is assessed what parts of the system are likely to be affected by a change on the related artifacts.

3. And last, once we know where have been affected the artifacts by what kind of changes, they will carry out some co-evolution actions. Sometimes, this co-evolution can be done automatically, and sometimes manually, depending on the complexity of the kind of impact. The aim of the co-evolution is to reestablish the synchronization between artifacts.

Change propagation [Men08] is necessary when a change to one part of the software system requires other system parts that depend on it to be changed as well.

3.3 Design Space of Synchronization

Synchronization (a.k.a. change propagation or co-evolution) is the process of enforcing consistency among a set of artifacts and *synchronizers* are procedures that automate (fully or in part) the synchronization process. Synchronizers can be classified along two dimensions: the artifacts that need to be synchronized and the solution to carry out that synchronization. In [AC07], they defined excellently a design space for synchronization. This design space gives the frame where some of the works in the Thesis rest. A summary of that design space will be given below.

3.3.1 Artifacts to be Synchronized

Artifacts are related to each other in different ways. For instance, a design model and its implementation code are related by *refinement* relationship.

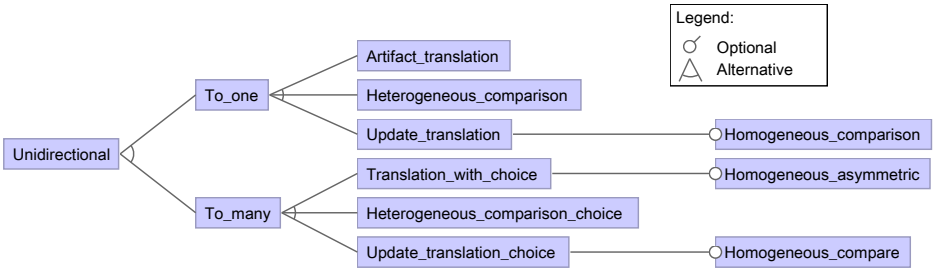

Figure 3.2: Unidirectional synchronizers

Also, a model and its metamodel are related by a *conformance* relationship. Some examples of relations among different kinds of artifacts are the following:

- The relation between a class diagram and KM3, which is *bijective*. KM3 is a textual notation that can be used for the specification of class diagrams. Artifacts expressed in one language can be translated into the other language without any loss of information.

- The relation between Java and type hierarchy, which is *functional*. Type hierarchy is an abstraction of a Java program because it contains a subset of the information contained in the program.

- The relation between a model and a metamodel, which is a *general relation*. Many models can conform to a single metamodel and a single metamodel can conform to many metamodels.

3.3.2 Categories of Synchronizers:

There are three categories of synchronizers, which are explained below:

- Unidirectional (in Figure 3.2): They synchronize in one direction at a time, meaning that they are most useful if only one of the artifacts was changed since the last synchronization. Depending on the cardinality of the end of the relation, they can be:

  - to-one: the mapping between source and target is a function from source to target.
    * Artifact translation: translates an entire source artifact into a consistent target artifact.
    * Heterogeneous artifact comparison: directly compares two artifacts of different types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact.
    * Update translation: translates an update to the source artifact into a consistent update of the target artifact. Optionally, the transform may use *homogeneous artifact comparison* to compute the source update as a difference between the original and the new source.
  - to-many: the relation is either a function in the target-to-source direction or a general relation. Consequently, the mapping may specify several alternative target artifacts that a synchronizer could return for a given source artifact. It will require a mechanism for selecting one target artifact from the set of possible alternatives.
    * Artifact translation with choice: to translate the new source into a set of possible new targets and selects one target using a decision function. Optionally, it can be used *homogeneous asymmetric artifact merge with choice* to merge the selected new target with the original target.
    * Remaining synchronizers (heterogeneous comparison and update translation) work similarly to their to-one counterparts.

- Bidirectional (in Figure 3.3): They involve propagating changes in both directions. They can also be used when both artifacts have

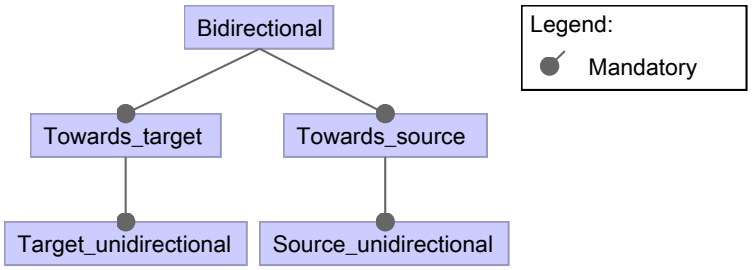

Figure 3.3: Bidirectional synchronizers

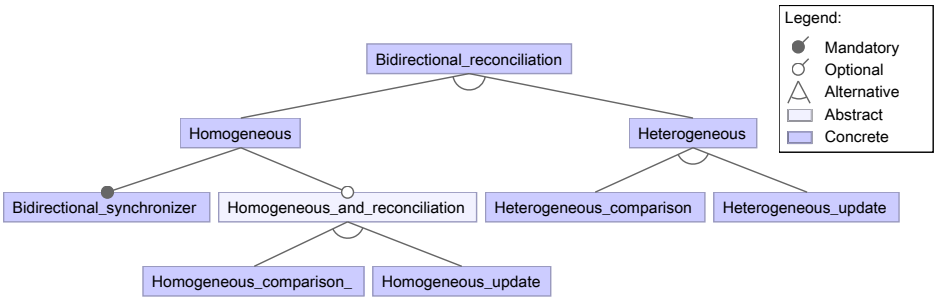

Figure 3.4: Bidirectional synchronizers with reconciliation

changed since the last synchronization. However, they cannot be used to resolve conflicting changes to both artifacts, as one artifact acts as a slave and its changes may get overridden. They can be seen as a pair of unidirectional synchronizers, one for each direction.

  - Towards target: represents the unidirectional synchronizer from source to target.
  - Towards source: represents the unidirectional synchronizer from target to source.

- Bidirectional with reconciliation (in Figure 3.4): They can be used to synchronize both artifacts at the same time. Thus, these synchronizers are also applicable in situations where both artifacts were changed since the last synchronization and they can be used for conflict resolution in both directions.

- Homogeneous reconciliation: the new source artifact or the update of the source artifact need to be first translated into the target type. Depending whether the entire artifact or just the update is translated, the comparison and reconciliation is done either by *homogeneous artifact comparison and reconciliation with choice* or its *update.*
- Heterogeneous reconciliation: implies a heterogeneous comparison between the artifacts or the updates. It can be implemented using the operator *heterogeneous artifact comparison and reconciliation with choice* or its *update.*

3.4 Metamodel Evolution

As any other software artifact, metamodels are subject to evolution. During design alternative metamodel versions may be developed. During implementation, metamodels may be adapted to a concrete metamodel formalism supported by a tool. Finally, during maintenance, errors in a metamodel may be corrected. Moreover, parts of the metamodel may be redesigned due to a better understanding or to facilitate reuse [Wac07]. These factors can be considered in the context of software maintenance.

An interesting description of the evolution of a real metamodel (GMF) is given in [HRW09]. They summary the type of changes affecting the metamodel during many releases of its life cycle.

3.4.1 Metamodel-Model Co-evolution

One of these problems that has been widely studied is the co-evolution (a.k.a. adaptation, migration or synchronization) of model instances to metamodel evolution. In some works, changes have been classified into: *additive*, that enlarge the modeling space of the language (more models are valid); *subtractive* changes, which reduce the modeling space, and *updative* changes, that ensure the same size.

There are some tools that assist in the co-evolution, and Table 3.1 summary them.

A brief explanation of the dimensions used in the table to characterize the works are the following:

- Differencing: Differences can be obtained from two versions of a metamodel (two "states"), but without the knowledge of what happened between them: state-based. On the other hand, there are some predefined operations that can be applied to a metamodel in order to evolve it: operation-based.

- Adaptations: If the metamodel changes are considered as a whole or if they are managed one at a time.

- Order: Migrations can be directly specified using a language provided by the considered technique (Single order), or they can be automatically generated starting from the metamodel changes (Higher-order).

- Transformation type: If the migration gives the adapted artifact as a new entity (Out-place) or as the initial element with the adaptation operating directly on it (In-place).

- Focus: If the migration approach is dedicated to a specific kind of artifact or if it is a general approach capable of migrating any artifact.

- Paradigm: If the adaptation logic is defined in a declarative or imperative manner.

- Concrete syntax: If the notation used for specifying migrations is textual or graphical.

- Lack of information management: Some adaptation cases cannot be done automatically and need further information. Sometimes that lack of information is solving asking for more information to the user

	EMFMigrate	**Cope/Edapt [Her11, HVW11]**	**Flock**[a]	**GMF Evolution**	**Garcés [GJCB09b]**
Differencing	State based	Operation based	Operation based	State based	State based
Adaptations	Batch	Interleaved	Batch	Batch	Batch
Order	Single order	Single order	Single order	Single order	Higher order
Transf. type	Out-place	In-place	Out-place	Out-place	Out-place
Focus	Any	Model	Model	Editor	Model
Paradigm	Declarative	Imperative	Imperative	Declarative	Declarative
Concrete syntax	Textual	Textual	Textual	Textual	Textual
Lack of information management	Heuristics based/ user specified	User specified	User specified	Heuristics based	Heuristics based
Modularity/ reuse	Yes	No	No	No	No
Refinement	Yes	No	No	No	No
Model navigation	OCL based	Groovy based	OCL based	OCL based	OCL based

Table 3.1: Tools for co-evolution (based on [RIP12a])

[a] **http://www.eclipse.org/epsilon/doc/flock/**

(User specified) or alternatively, the migration program can apply some heuristics.

- Modularity/reuse: If applied migration strategies can be applied to another scenarios.
- Refinement: If migration strategies can be refined.
- Model navigation: Language used to navigate and query artifacts to be adapted.

Some of the approaches have chosen to automate as much as possible the migration (e.g. Garcés), while others has proposed more manual and guided approaches (e.g. Flock).

3.4.2 MM-Editor Co-evolution

In the Eclipse Modeling Framework (EMF) [RIP12b, SBPM09] different approaches have been proposed to define concrete syntaxes of modeling languages, e.g. GMF[1] for developing graphical editors, EMFText[2], TCS [JBK06], and XText [EV06] for producing textual editors. Essentially, all of them are generative approaches able to generate working editors starting from source specifications that at different extent are related to the abstract syntax of the considered DSML, which often is a metamodel. When this metamodel defining the abstract syntax changes, the editor models are affected.

In [RIP13] they automate the propagation of metamodel changes to textual concrete specifications given in TCS tool.

Concerning the adaptation of GMF editors, in [RLP10] the authors introduce an approach to automate the propagation of domain-model changes (i.e., metamodel changes) to the EMFGen, GMFTool, and GMFMap models required by GMF to generate the graphical editor.

[1] http://www.eclipse.org/gmf/
[2] http://www.emftext.org/index.php/EMFText

3.5 Platform Evolution

Although evolution in MDE ecosystems (and its consequent co-evolutions) has been treated in several works [CREP08, IPM12], platform evolution and its impact on the ecosystem has been overlooked. A *platform* comprises a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns; any application supported by that platform can use it without concern for the details of how the functionality provided by the platform is implemented [MM03]. Platforms do evolve [DJ06] and changes in the platform might impact related artifacts. As a case in point, the developer might have to change protected areas in the code that do not work in the new platform, the model-to-text transformation that generated the code (and potentially many others) may need to be updated, and depending on how close to the platform it is, also the metamodel to which the domain model conforms. She needs to co-evolve those artifacts so that the code generated using them is adapted to the changes. Otherwise, the code generated after platform evolution could prove to be useless.

In platform evolution, the code generators and application framework change to reflect new requirements on the target platform. Some years ago, in [VWVDVD07] they stated that little or no support existed for platform evolution, and as far as I know there have not been any advances from that time.

Chapter 4

Model Transformation Co-evolution: a Semi-automatic Approach

4.1 Introduction

As any other software artifact, metamodels are subject to evolution. During design alternative metamodel versions may be developed. During implementation metamodels may be adapted to a concrete metamodel formalism supported by a tool. Finally, during maintenance, errors in a metamodel may be corrected. Moreover, parts of the metamodel may be redesigned due to a better understanding or to facilitate reuse [Wac07]. Simultaneously, metamodels lay at the very center of the model-based software development process. Both models and transformations are coupled to metamodels: models *conform to* metamodels, transformations *are specified upon* metamodels. Hence, metamodel evolution percolates both models and transformations. We focus on transformation co-evolution after metamodel evolution. This problem is illustrated in Figure 4.1.

We do not force to describe that metamodel evolution in terms of *ad-hoc* operands [Her11], but evolution is ascertained from differences

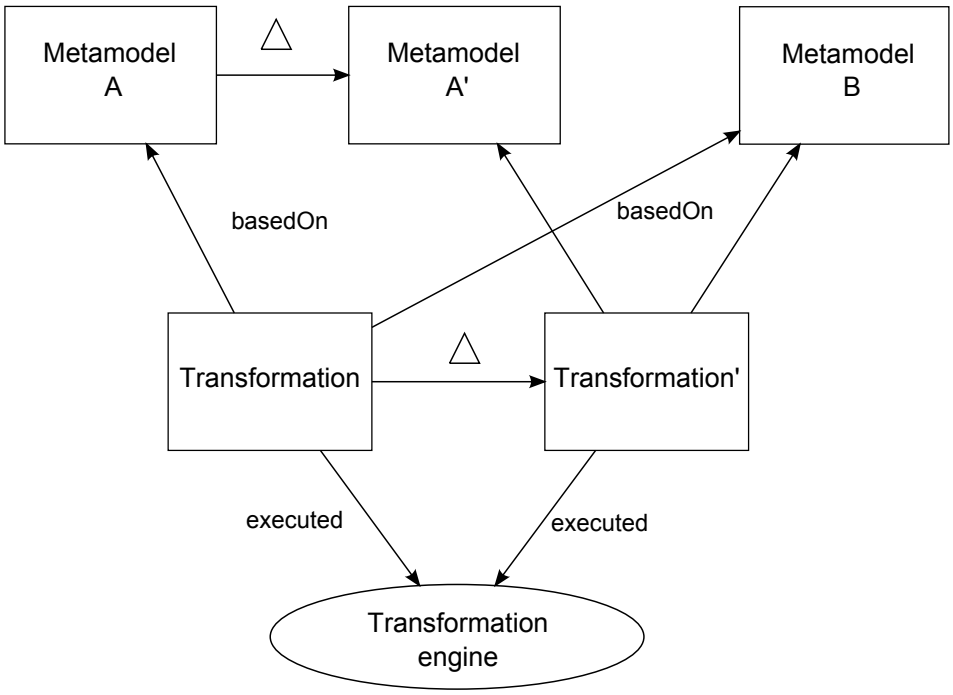

Figure 4.1: Metamodel evolution and transformation co-evolution

between the original and the evolved metamodel. Next, depending on their impact, differences are classified as [CREP08]: *(i) Non Breaking Changes (NBC)*, i.e., changes that do not affect the transformation; *Breaking and Resolvable Changes (BRC),* i.e., changes after which the transformations can be automatically co-evolved; and *Breaking and Unresolvable Changes (BUC),* i.e., changes that require human intervention to co-evolve the transformation. Finally, the transformation is subject to distinct actions based on the type of the change, i.e., no action for NBC, automatic co-evolution for BRC, and assisting the user for BUC. The outcome is an evolved transformation that tackles (or warns about) the evolved metamodel. This approach is realized in a prototype that takes as input the original Ecore metamodel, the evolved Ecore metamodel and an ATL transformation [JABK08], and outputs an evolved ATL transformation. It achieves the automatic co-evolution of BRC and the assistance for BUC. The prototype makes intensive use of HOTs whereby the original

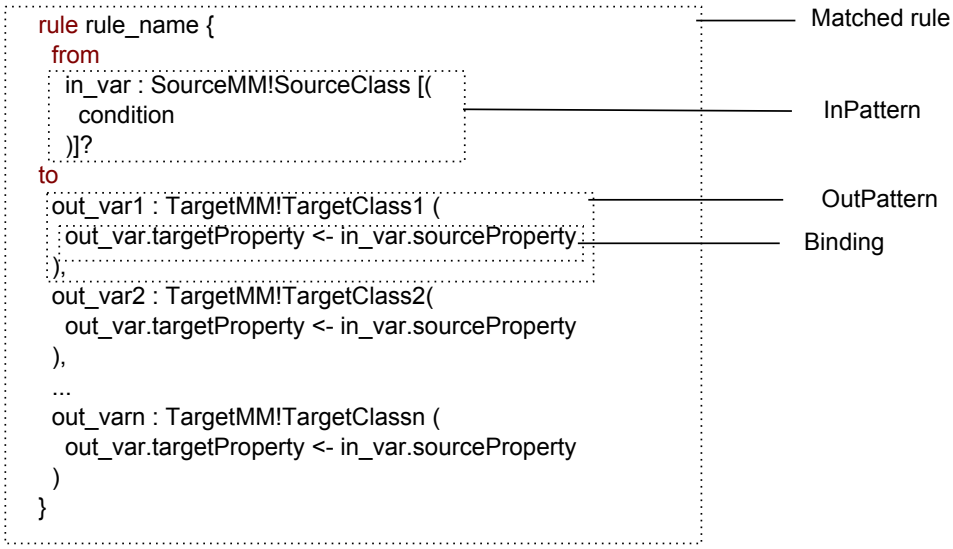

Figure 4.2: Sample ATL transformation

transformation is handled as a model which needs to be mapped into another model (i.e. the evolved transformation). The approach can be generalized to any transformation language that provides a metamodel representation.

This chapter starts with a motivating scenario. Next, it is outlined the co-evolution process whose two main stages, detection and co-evolution, are presented in more detail in Subsections 4.4 and 4.5, respectively. Subsection 4.6 introduces the prototype architecture and describes one of its HOT rules. Related work and conclusions end the chapter.

4.2 Motivating Scenario

As ATL will be used as transformation language, and ATL code snippets will be shown, basic concepts of the language are shown in Figure 4.2. The main structures to specify the way target model elements are generated from source model elements are matched rules. The rule is composed of a *from* part (a.k.a *left* or *source* part) and a *to* part (a.k.a. *right* or *target* part) that match source metamodel and target metamodel elements. In the

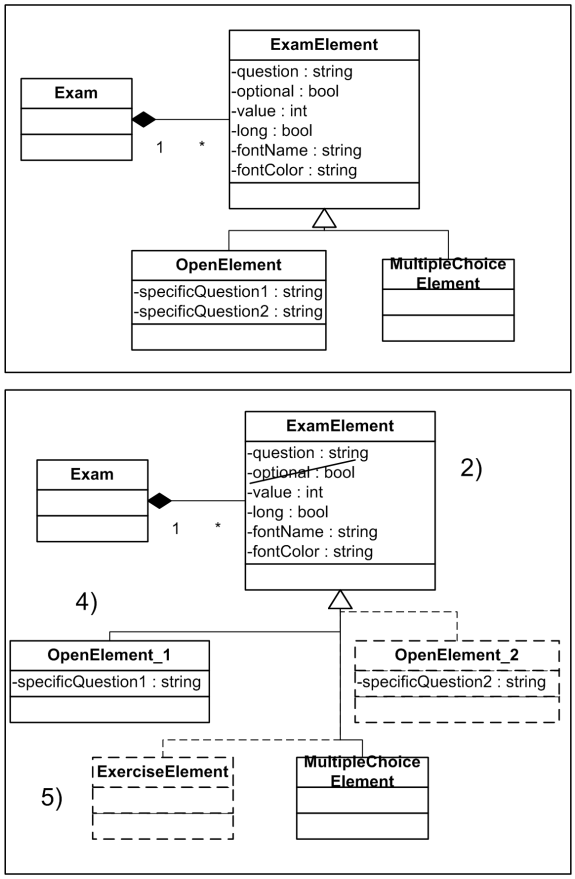

Figure 4.3: *ExamXML* metamodel evolution: original (above) and evolved (below)

from part, it can be specified a condition (a.k.a. filter) that filters the source elements. In the *to* part, there can be several OutPatterns, as one source element can be mapped to several target elements. The bindings will be in charge of matching properties.

```
1 module exam2mvc;
2 create OUT : AssistantMVC from IN : ExamXML;
3 rule Exams {
4   from xml : ExamXML!Exam (
5     examItems <- xml.elements)
6 }
7 rule OpenQuestion {
```

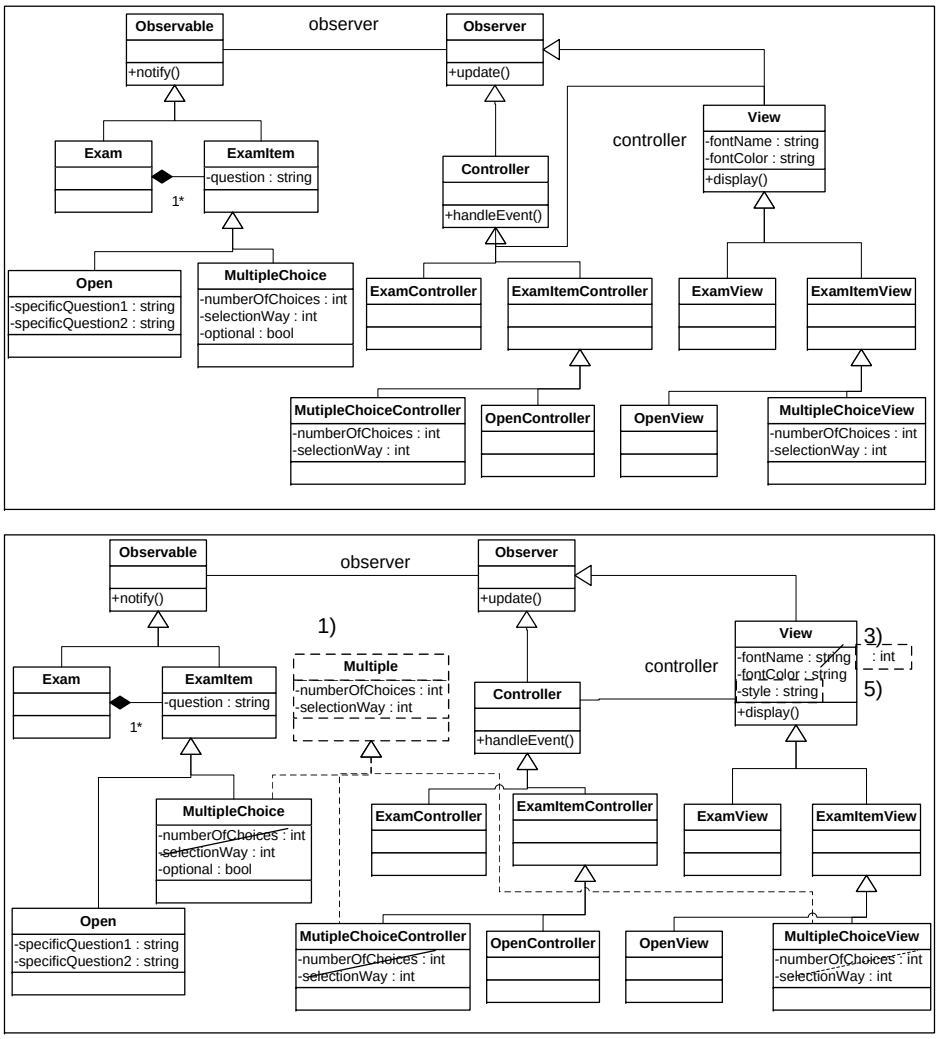

Figure 4.4: AssistantMVC metamodel evolution: original (above) and evolved (below)

```
8   from xml: Exam!OpenElement
9   to controller : AssistantMVC!OpenController(),
10    view : AssistantMVC!OpenView(
11      controller <- controller,
12      fontName <- 'Times',
13      fontColor <- 'Red'),
14    model : AssistantMVC!Open (
15      question <- xml.question,
16      specificQuestion1 <- xml.specificQuestion1,
17      specificQuestion2 <- xml.specificQuestion2,
18      observers <- view)
19 }
20 rule MultipleChoice {
21   from xml : ExamXML!MultipleChoiceElement
22   to controller : AssistantMVC!MultipleChoiceController(),
23     view : AssistantMVC!MultipleChoiceView(
24       controller <- controller,
25       fontName <- 'Times',
26       fontColor <- 'Red' ),
27     model : AssistantMVC!MultipleChoice (
28       question <- xml.question,
29       observers <- view )
30 }
```

Listing 4.1: Exam2MVC transformation

We use the popular *Exam2MVC* transformation [Kur05] as a running example. This scenario envisages different types of exam questions from which Web-based exams are automatically generated along the MVC pattern [Kur05]. Figures 4.3 and 4.4 present the *ExamXML* and the *AssistantMVC* metamodels. The *Exam2MVC* transformation (Listing 4.1) generates an *AssistantMVC* model out of an *ExamXML* model.

Next, we introduce a set of evolution scenarios to be considered throughout the section (see Figures 4.3 and 4.4):

- Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This new class abstracts away the commonality of three existing classes: *MultipleChoiceController,*

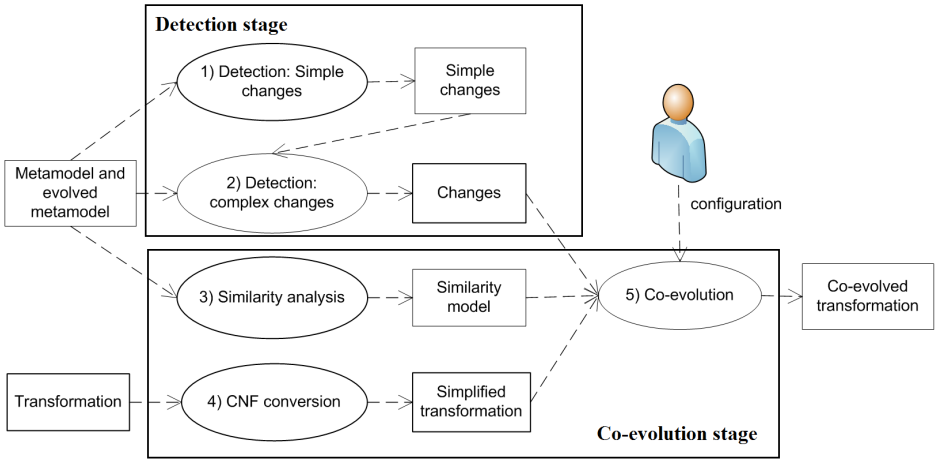

Figure 4.5: Transformation co-evolution process.

MultipleChoiceView and *MultipleChoice*.

- Scenario 2. *optional* property is deleted from *ExamXML's ExamElement*.
- Scenario 3. The *AssistantMVC's fontColor* metaproperty is changed from *string* to *integer*.
- Scenario 4. The *ExamXML's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2*.
- Scenario 5. *ExerciseElement* subclass is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass.

Now the question is how these changes impact the *Exam2MVC* transformation, better said, how can the designer be assisted in propagating these changes to the transformation counterpart. Next subsection outlines the process.

4.3 Transformation Co-evolution Process: An Outline

```
1  rule OpenQuestion {
2    from xml: Exam!OpenElement
3    to controller : AssistantMVC!OpenController(),
4      view : AssistantMVC!OpenView(
5        controller <- controller,
6        fontName <- 'Times',
7        fontColor <- 'Red'),
8      model : AssistantMVC!Open (
9        question <- xml.question,
10       specificQuestion1 <- xml.specificQuestion1,
11       specificQuestion2 <- xml.specificQuestion2,
12       observers <- view)
13 }
14             | |
15            _| |_
16            \   /
17             \ /
18 --SPLITTED RULE 1
19 rule OpenQuestion{
20   from xml : ExamXML!OpenElement_1
21   to controller : AssistantMVC!OpenController,
22     view : AssistantMVC!OpenView (
23       controller <- controller,
24       fontName <- 'Times',
25       fontColor <- 'Red' ),
26     model : AssistantMVC!Open (
27       question <- xml.question,
28       specificQuestion1 <- xml.specificQuestion1,
29       observers <- view)
30 }
31 --SPLITTED RULE 2
32 rule OpenQuestion2{
33   from xml : ExamXML!OpenElement_2
34   to controller : AssistantMVC!OpenController,
35     view : AssistantMVC!OpenView(
36       controller <- controller,
37       fontName <- 'Times',
38       fontColor <- 'Red'),
```

```
39      model : AssistantMVC!Open(
40        question <- xml.question,
41        specificQuestion2 <- xml.specificQuestion2,
42        observers <- view )
43  }
```

Listing 4.2: *Exam2MVC* transformation: original -above-; co-evolved -below-.

This section outlines the transformation co-evolution process aiming at assisting designers by automating co-evolution whenever possible. This process comprises two main stages: *detection* and *co-evolution* (see Figure 4.5). Inputs include the original metamodel (M), the evolved metamodel (M') and the original transformation (T).

Detection stage. The original metamodel and the modified metamodel are compared, and a set of differences are highlighted. Differences can range from simple cases (e.g. 'class renaming') to more complex ones (e.g. 'class splitting'). Simple changes are those that are conducted as a single shot by the user. By contrast, complex changes are abstractions over simple ones as they conform a meaningful transaction on the metamodel. Complex changes need to be treated as a unit not only from the perspective of the metamodel, but also from the co-evolution perspective. Otherwise, we risk to miss the intention of the designer when evolving the metamodel, and hence, to propagate this misunderstanding to the transformation. To this end, the detection stage includes two tasks: simple-change detection and complex-change detection. The outcome is a set of changes, both simple and complex.

Co-evolution stage. Having a set of metamodel changes as input, this step first classifies changes based on their impact on the transformation rules. Based on the notation used in [CREP08], we identify three types of metamodel changes:

1. *Non Breaking Changes (NBC).* These changes have no impact on the transformation. This case is illustrated by the first scenario: the introduction of the *Multiple* class as an abstraction of two existing

classes. *Superclass extraction* has generally no impact on the transformation since metaclass properties are still reachable through inheritance. Therefore, this type of changes need to be detected, but no further action is required.

2. *Breaking and Resolvable Changes (BRC).* These changes do impact the transformation rules, but this impact is amenable to be automated. The fourth scenario is a case in point. Here, *OpenElement* is splitted into *OpenElement_1* and *OpenElement_2* classes. Accordingly, rules having *OpenElement* as its source might give rise to two distinct transformation rules that tackle the specifics of *OpenElement_1* and *OpenElement_2* (see Listing 4.2*).* Another example is the removal of the *optional* metaproperty from the *ExamElement* class (see *change 2* on the bottom of Figure 4.3). Metaproperty removal is an example of an BRC scenario: transformation rules that refer to the removed metaproperty no longer work. At the same time, it suffices to delete any reference to the deleted metaproperty from the transformation to produce a transformation that corresponds to the evolved metamodel. Hence, as the transformation can be co-evolved automatically, metaproperty removal is classified as resolvable.

3. *Breaking and Unresolvable Changes (BUC).* These changes also impact the transformation, but full automatization is not possible and user intervention is required. Reasons include: the semantics of the metamodel, the specific characteristics of the transformation language, or the specificity of the change. Hence, it will be designer's duty to manually guide the co-evolution. This is illustrated by scenario 3: *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. Type changes are the most ambiguous ones due to transformation languages being dynamically typed, and hence, susceptible to generate type errors at runtime. For instance, a rule could assign '*Times*' to *fontName*. *FontName* has

now be turned into an integer, hence, making this rule inconsistent. In those cases, the option is to warn about the situation, and let the designer provide a contingency action (e.g. coming up with the *"integer"* counterpart of the formerly valid value '*Times*').

In short, for each type of change (i.e. NBC, BRC or BUC), it is proposed a course of action: no action, automatic transformation, and assisted transformation, respectively. To this end, the co-evolution process is complemented by two auxiliary steps: a **Conversion to Conjunctive Normal Form (CNF) step** (to address removals) and an optional **similarity analysis step** (to handle additions). Next two subsections delve into the details.

4.4 Detection Stage

This stage takes as input both the original metamodel and the evolved metamodel, and infers the set of changes that went in between. Metamodel evolution will not be forced to be described in terms of predefined operands, but comparing two states of the metamodels without having the evolving process between the two versions. This is useful, as often a person that uses a metamodel is not in charge of it, and evolution is generally out of its control. This is achieved through two tasks: simple-change detection and complex-change detection.

4.4.1 Simple-Change Detection

We detect simple changes as a difference between the original metamodel and the evolved metamodel. To this end, we use *EMF Compare* [Tou06]. This tool takes two models as input and obtains the differences along the *Difference metamodel*. Back to our running example, EMF Compare is used to detect the simple changes between the original and evolved *ExamXML* metamodel as well as the original and evolved *AssistantMVC* metamodel. The output is a *Difference* model. Figure 4.6 (above)

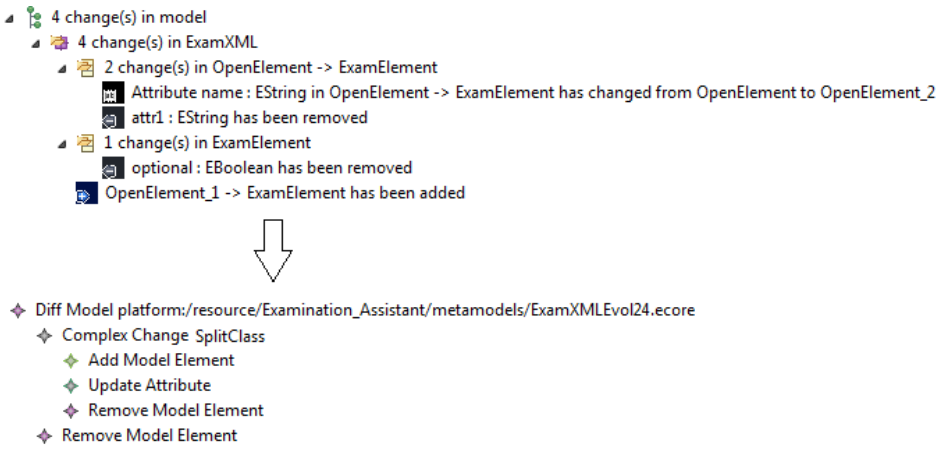

Figure 4.6: The *Difference* model (above) *& DiffExtended* model (below) for the running example.

illustrates this *Difference* model for the *ExamXML* metamodel (scenarios 2 and 4): *UpdateAttribute*, *RemoveModelElemen*t, *AddModelElement* and *RemoveAttribute*. In other words, it detects that the name of the class is changed from *OpenElement* to *OpenElement_2*, the *specificQuestion1* metaproperty is removed from *OpenElement_2*, the attribute *optional* is being removed, and a new class with name *OpenElement_1* is added. Simple changes that account for a more abstract complex change are arranged as descendants of the complex change (e.g. *AddModel*, *UpdateAttribute*, *RemoveElement* are now part of a *ComplexChange* whose *changeType* is *SplitClass*).

4.4.2 Complex-Change Detection.

Simple changes might be semantically related to achieve a common higher-order modification. For a list of complex changes refer to [HVW11] (we are going to analyze those relevant from the point of view the transformation co-evolution). For instance, the previous *AddModelElement* simple change hides a class split. We need to infer that a set of simple changes unitedly account for a split. Alternatively, we

Figure 4.7: *DiffExtended* metamodel: *EMFCompare*'s *Difference* metamodel is extended with the *ComplexChange* class.

risk to treat each simple change on its own, which could lead to unwanted co-evolution in the transformation.

We regard complex changes as predicates over simple changes. *C* is the set of metaclasses and *P* the set of metaproperties of a metamodel. Auxiliary predicates needed to define them are defined in Table 4.1.

Auxiliary predicate	**Description**
Subclass(s∈ C, c∈ C)	*s* is subclass of *c*
Added_class(c∈ C)	*c* is added to the metamodel
Added_attribute(p∈ P, c∈ C)	*p* has been added to *c*
Deleted _attribute(p∈ P, c∈ C)	*p* has been deleted from *c*
IsAttributeOfClass(p∈ P, c∈ C)	*p* belongs to *c*
Added _supertype(s∈ C, c∈ C)	supertype relationship has been added from *s* to *c*
Deleted_supertype(s∈ C, c∈ C)	supertype relationship has been deleted from the *s* to *c*
Added_reference(p∈ P, c∈ C, d∈ D)	Reference *p* from *c* to *d* is added
Deleted_reference(p∈ P, c∈ C, d∈ D)	Reference *p* from *c* to *d* is deleted
Added_composition(s∈ C, c∈ C)	composition relationship has been added from the *s* to the *c*
Composed_name(z: string, p: string, x: string)	delivers a new string *x* out of input strings *z* and *p*
Splitted_name(c, x)	returns true if *c* can be obtained from *x* by concatenating such suffix. A notation convention exists to name split classes: the name of the original class concatenated with a number (e.g. *OpenElement_1, OpenElement_2*)
SplitClassName(c∈ C)	returns true if the new name of c is the concatenation of the old name and "_1"

Table 4.1: Auxiliary predicates used to define complex changes

The list of detection predicates is defined in Table 4.2.

Table 4.2: Complex changes

Complex change	**Detection predicate**
ExtractSuperclass (c∈C)	iff Added_class(c) ∧ ∃p∈P, ∃s∈C (Added_attribute(p, c) ∧ Added_supertype(s, c) ∧ Deleted _attribute(p, s))
PullMetaproperty (c∈C, s∈C)	iff ∃p∈P (Subclass(s, c) ∧ Added_attribute(p,c) ∧ Deleted _attribute(p, s))
PushMetaproperty (p∈ P)	iff ∃s, c∈C (Subclass(s, c) ∧ Deleted_attribute(p,c) ∧ Added _attribute(p, s))
FlattenHierarchy (c∈C)	iff (Deleted_class(c) ∧ ∀p∈ P ǀ IsAttributeOfClass(p, c) , ∀s∈ C ǀ Subclass(s, c) (Deleted _attribute(p,c) ∧ Deleted_supertype(s, c) ∧Added_attribute(p, s)))
MoveMetaproperty (c∈C, p∈P, d∈C)	iff (Deleted_attribute(p,c) ∧ Added _attribute(p, d))
ExtractMetaclass (c∈C, d∈C)	iff (Added_class(d) ∧ ∀p∈P ǀ IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
InlineMetaclass (c∈C, d∈C)	iff (Added_class(d) ∧ Deleted_class(c) ∧ ∀p∈ P ǀ IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
InheritanceTo Composition(c∈C, d∈C)	iff (Deleted_supertype(d, c) ∧ Added _composition(d, c))

Generalize Supertype (c∈C, s∈C, d∈C)	iff (Deleted_supertype(d, s) ∧ Added _supertype(d, c) ∧Subclass(s, c))
InlineSubclass(c∈C, d∈C)	iff (Deleted_class(c) ∧ Subclass(c, d) ∧ ∀p∈P | IsAttributeOfClass(p, c) (Added _ attribute(p,d) ∧ Deleted_attribute(p, c)))
ReferenceTo Identifier(c∈C, d∈C, p∈P)	iff (Deleted_reference(p, c, d) ∧ Added _attribute(p, c) ∧ Added_attribute(p, d))
SplitReference ByType(c∈C, d∈C, x∈C, s∈C, p∈P, y∈P, z∈P)	iff (Deleted_ reference(p, c, d) ∧ Added_reference(y, c, x) ∧ Added_reference(z, c, s))
PropertyMerge(p∈P, z∈P, x∈P)	iff ∃c∈C (Deleted_attribute(p,c) ∧ Deleted _attribute(z, c) ∧ Added_attribute(x, c) ∧ Composed_name(z, p, x)). The last predicate delivers *x* by concatenating strings *z* and *p*.
ClassMerge(c∈C, d∈C)	iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Deleted_class(d) ∧ Composed_name(c, d, x))
SplitClass(c∈C, d∈C, x∈C)	iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Added_class(d) ∧ Splitted_name(d, c)∧ SplitClassName(c)).

Implementation wise, simple changes are obtained using *EMFCompare* using the *Difference* metamodel. We propose to extend the *Difference metamodel* to account also for complex changes. Figure 4.7 shows an extract of the *DiffExtended* metamodel. Using the predicates aforementioned we infer complex changes that are represented as a *DiffExtended* model. Figure 4.6 provides a *DiffExtended* model where complex changes are also introduced. In the case that a simple change can belong to different complex changes, the biggest one has priority, e.g. *FlattenHierarchy* over *MoveMetaproperty*, as the first one includes the

second.

In short, this task is realized as a transformation that takes a *Difference* model as input and obtains a *DiffExtended* model that includes both single and complex changes. Now, we are ready to percolate those changes to the transformation rules.

4.5 Co-evolution Stage

After detecting the changes, they have to be propagated to the transformation. This co-evolution process must satisfy these requirements: (based on [vdBPV11])

1. The co-evolved transformation must be syntactically correct, i.e., conform to the transformation metamodel.

2. Only the minimal necessary changes must be carried out in the co-evolution process.

3. The semantic of the transformation must be preserved.

4. The process must maximize automation, minimizing user intervention.

4.5.1 Similarity Analysis Step

Additional degrees of automatization can be achieved by using metamodel matching techniques. Metamodel matching is a generic method that infers the mapping specification taking advantage of similarities between source and target metamodels. Its performance will be proportional to similarity degree between metamodels. The similarity analysis is conducted between the source and target metamodels using tools such as AML (AtlanMod Matching Language) [GJCB09a]. These tools compute similarity based on the element names and the structural similarity of the

Transformation	**Element-level**			**Structure-level**		
	Equal	*Similar*	*%*	*Mapped source element*	*B*	*C*
XSLtoQuery	31	16	51	7	2	0
KM32Measure	62	11	17	4	0	0
UMLActivityDiagram	7	2	28	-	-	-
TableToExcel	6	0	0	3	3	0
SwQuality2Mantis	38	9	23	2	1	0
SwQuality2Bugzilla	26	2	8	-	-	-
RSS2Atom	20	7	35	3	1	0
PathExp2Petrinet	16	4	25	3	2	0
MsOffice2Quality	24	3	12	4	0	0
Class2Relational	22	6	27	3	2	0
MetaHAcme	50	16	32	-	-	-
Measure2Table	7	2	29	1	0	0
Make2Ant	16	4	25	4	0	0
KM32Metric	27	0	0	2	0	0
Bibtex2DocBook	26	2	8	5	0	1
KM32Dot	75	17	23	4	0	1
Java2Table	7	0	0	2	0	0

Table 4.3: Metamodel matching success rate

metamodels. The output can be used to assist designers to fill the gaps. The approach rests on the matching effectiveness. To ascertain an average effectiveness, we performed an empirical experiment based on a test-bed of 17 transformations from the ATL zoo[1] for the following heuristics:

- Element-level techniques compute mapping elements by analyzing entities in isolation, ignoring their relations with other entities. This seems to be useful in the case of adding a metaproperty, to measure its similarity with the metaproperties of the metaclass it is mapped to. In this case an experiment is done in the test bed to check if in the mappings between metaproperties there is textual similarity. The

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

results are the following: in a total of 460 bindings, the number of equal or similar metaproperties were 101 (22%).

- Structure-level techniques compute mapping elements by analyzing how entities appear together in a structure. The experiment consisted on computing the number of elements that are categorized as similar by the matching algorithm that are really mapped together. Being *A* the false negatives (matches needed but not identified), *B* the true positives (matches needed and which have also been correctly matched by the automatic match operation) and *C* the false positives (matches falsely proposed by the automatic match operation), these were the results: A total = 37 (77%), B total = 11 (23%) and C total = 2 (4%). As can be deducted from the data, few cases are detected by metamodel matching, but false positives are low, so they are reliable.

As we can see in the data from the Table 4.3, in both transformation patterns and metamodel matching, positive and negative cases are not distributed in the same way in the transformations. For instance, we can see in Table 4.3 that element-level similarity can be quite useful in *XSLtoQuery* case but is not useful at all in *KM32Metrics* transformation. Matching effectiveness had an average of 22-23% success (i.e., cases were an adequate binding could be suggested to the designer).

This step is optional. The developer might use it to infer mappings for additive evolution of metamodels. For this purpose, the weaving similarity model can be added as an input to the adaptation. This way, adaptations of additive cases not only will generate the skeleton of the rule or binding, but they will complete the skeleton with infered information.

4.5.2 Conjunctive Normal Form Conversion Step

Rule filters are first-order predicates, normally specified using OCL. Equivalence rules of Predicate Calculus are applied to each boolean expression to get its equivalent *Conjunctive Normal Form (CNF)*, i.e., a

conjunction of clauses, where a clause is a disjunction of literals. A *literal* is any subset of a constraint that can be evaluated to a boolean value and does not include a boolean operator (or, xor, and, not and implied). Once in CNF, filters can be subject to "surgically removal", as explained in Subsection 4.5.4. The translation of logical formulas to a CNF is specified in [CC04] (see the reference for further details):

1. Eliminate non-basic boolean operators (if-then-else, implies, xor), using the rules:
   (a) A implies B == not A or B
   (b) if A then B else C == (A implies B) and (not A implies C) == (not A or B) and (A or C)
   (c) A xor B == (A or B) and (not A or not B)
2. Move not inwards until negations be immediately before literals by repeatedly using the laws
   (a) not (not A) == A
   (b) DeMorgan's laws: not (A or B) == not A and not B not (A and B) == not A or not B
3. Repeatedly distribute the operator or over and by means of:
   (a) A or (B and C) == (A or B) and (A or C)

Once expressions have been translated into CNF, their literals are only connected by *and*, *or* and *not* operators. Next, literals that reference to an element to be removed are marked as undefined, as it cannot be evaluated. Next, to know which parts of the expression need to be deleted, the following rules are applied:

1. literal AND undefined -> literal
2. literal OR undefined -> undefined
3. NOT undefined -> undefined

4.5.3 Co-evolution Step

We treat transformations as models. That is, transformations are described along a transformation metamodel. Therefore, it is possible to define HOTs that take a transformation as input, and return a somehow modified transformation. This is precisely the approach: define correspondences that map the original transformation into an evolved transformation, taking the changes obtained during the detection stage as parameters. These HOTs are realized as ATL rules. In what follows, we summarize those rules in terms of co-evolution actions. These actions are expressed as predicates over the original transformation rules. To this end, we capture a transformation rule R as a tuple *Rule(id, source, targets, filters, mappings)* where *"source"* and *"targets"* refer to classes of the input and output metamodel, respectively; *"filters"* is a set of related predicates over the source element, such that the rule will only be triggered if the condition is satisfied; finally, *"mappings"* refer to a set of bindings to populate the attributes of the target element. A binding construct establishes the relationship between a source and a target metamodel elements. Normally, a mapping part contains a binding for each target metaclass' property. Its semantics denote what needs to be transformed into what instead of how the transformation must be performed. The left-hand side must be an attribute of the target element metaclass. The right-hand side can be a literal value, a query or an expression over the source model. Figure 4.1 illustrates an example of a transformation in ATL.

Transformation rules are the facts. Next, co-evolution actions are described through a set of operands and predicates over these rule facts. To avoid cluttering the description with iterations, we consider multi-valued predicates to return a single value. For instance, if a set of rules is used as parameter in the following *Bindings(r)*, bindings of all the rules in the set will be returned. Underscore will be used similarly to Prolog, as "don't care" variables. Predicates are intensional definitions of rule sets, and include: **RulesBySource(s)** denotes the set of rules whose source is *s;*

RulesByTarget(t) denotes the set of rules whose target is *t;* **Binding(r, p)** returns the bindings of rule *r* which hold property *p;* **Bindings(r)** returns the bindings of rule *r;* **TargetsOfRule(r)** returns the targets of rule *r;* **FiltersOfRule(r)** returns the filters of rule *r;* **FiltersOfProperty(p)** returns the filters where the property *p* appears.

Operands act on rules: **deleteRule(r)**, which deletes the rule *r;* **deleteTarget(r, t)** which deletes target *t* from rule *r;* **deleteBinding(r, b)**, which deletes binding *b* from rule *r;* **addRule(r)**, which adds rule *r;* **addTarget(r, t)**, which adds target *t* to rule *r;* **addBinding(r, b)**, which adds a binding *b* to rule *r;* **moveTarget(r1, t, r2)**, which moves *r1*'s target *t* together with its bindings to rule *r2;* **moveBinding(r1, b, r2)** which moves *r1*'s binding *b* to *r2*, provided *r2* holds a target that matches *b*'s lefthand side; **updateFilter(r1, f1, f2)**, which updates *f1* by *f2* among *r1*'s filters; **deleteFilter(r1, f1)**, which deletes one of *r1*'s filters; **updateBinding(r1,b1, b2)**, which substitutes *r1*'s binding *b1* by *b2;* **updateSource(r, s1, s2)**, which updates source *s* of rule *r* to *s2;* **concatClass(c1, c2)**, which concatenates two classes names. These operands are used to specify how metamodel changes impact the transformation rules, i.e. the co-evolution actions. Tables 4.4 and 4.5 summarize the actions related to simple and complex changes, respectively.

Table 4.4: Adaptation to simple changes

Simple change	Type	Adaptation
removeMetaclass (c∈ C)	BRC	deleteRule(RulesBySource(c)), deleteFilter(RulesBySource(c), FiltersOfProperty(c.properties)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), c.properties))

remove Metaproperty (p∈ P)	BRC	deleteFilter(RulesBySource(c), FiltersOfProperty(p)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), p)). Deletions should be minimal (in Subsection 4.5.4)
updateLowerBound (p∈ P, NewBound)	BRC	updateFilter(_, FiltersOfProperty(p), f2), updateBinding(_, Binding(_, p), b2). In case *lowerBound* converts from 1 to *, *f2* will insert a *forAll* expressions to check that all instances fulfill the condition and *b2* will use the *first()* to take the first element of the sequence. In case *lowerBound* changes from * to 1, *asSequence()* will be used in *f2* and *b2* to convert an element into a sequence.
updateUpperBound (p∈ P, NewBound)	NBC	No action
updateEType	BUC	Syntactically right, but possible runtime type errors (refer to [SJ07]). A warning note is generated.
updateESuperTypes	BUC	(if a metaproperty of the ancestors is accessed) Propose to copy the metaproperty of the superclass in the class.

updateIsAbstract (c∈ C,NewValue)	NBC or BRC	If *c* metaclass is turned into abstract (NewValue = "true") : Delete (Rule(c)), Delete (RHS(c)). If *c* metaclass is turned into a non-abstract class (NewValue = "false") then, do nothing.
updateELiterals (c∈ C)	BUC	Comment the structure where the literal is used, in case the user wants to use another literal. Alternative: use the default one.
addClass(c∈ C)	NBC	see Subsection 4.5.5
add Metaproperty (p∈ P)	NBC	see Subsection 4.5.5

Next, we illustrate the distinct casuistic using our running example, which was explained in Subsection 4.2:

- Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This is a NBC scenario.
- Scenario 2. The property *"optional"* is deleted from *AssistantMVC's ExamElement.* When a property is removed from the metamodel, different approaches can be taken, where the most simplistic one could be to remove the whole transformation rule where the property is used in a binding or boolean expression. However, this is a very restrictive and rather coarse-grained approach. We advocate the use of what we call the *principle of minimum deletion*, where only the part that is absolutely necessary is removed (see next subsection).
- Scenario 3. The *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. This is a BUC case.
- Scenario 4. The *AssistantMVC's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2*. As a result, rules having OpenElement as source should be co-evolved (see Figure 4.2). This

is the case of the *OpenQuestion* rule, which is splitted in two rules: *OpenQuestion_1 and OpenQuestion_2*. The former contains the bindings related to *OpenElement_1* while the latter keeps the bindings for *OpenElement_2*.

- Scenario 5. New subclass *ExerciseElement* is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass. Additive evolution is a NBC case. Even though, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose we include in the co-evolution the option to generate partially new rules, as they are not fully automatable (see Subsection 4.5.5).

Table 4.5: Adaptation to complex changes

Complex change	**Adaptation**
MoveMetaproperty (c∈ C, p∈ P, d∈ C) if c, d ∈ SourceClasses	updateBinding(RulesBySource(c), Binding(RulesBySource(c), p), newBinding(p)) where newBinding works out a binding by navigating to the new location of the property, in case both classes *c* and *d* are related (navigability exists through associations). If they are not related, user assistance will be needed.
MoveMetaproperty (c∈ C, p∈ P, d∈ C) if c, d ∈ TargetClasses	deleteBinding(RulesByTarget(c), Binding(RulesByTarget(c), p)) or if Binding(RulesByTarget(d), p) > 0: moveBinding(RulesByTarget(c), Binding (RulesByTarget(c), p), RulesByTarget(d)).

FlattenHierarchy (c∈ C) if c ∈ SourceClasses	deleteRule(RulesBySource(c)) and {if RulesBySource(Subclass(c)) >0 then moveBinding(RulesBySource(c), Binding(RulesBySource(c), p), RulesBySource (subclass(c))) else addRule(rule(_, subclass(c), _, _, _))}.
ExtractMetaclass (c∈ C, d∈ C) if c, d ∈ SourceClasses	addRule(rule(id, c, d, _, _)) and moveBindings(RulesBySource(c), Bindings(RulesBySource(c)), id).
ExtractMetaclass (c∈ C, d∈ C) if c, d ∈ TargetClasses	addTarget(Rule(c, d), d) and moveBinding(RulesBySource(c), Bindings(RulesBySource(c)), addTarget(Rule(c, d), d)).
InlineMetaclass (c∈ C, d∈ C)	”Extract metaclass” case and deleteRule(RulesBySource(c)).
InheritanceTo Composition (c∈ C, d∈ C)	When *c* is the source: updateFilter(RulesBySource(c), FiltersOfRule(RulesBySource(c)), f2), where in f2 *refImmediateComposite()* will be used in the filter. For instance: select(v | v.oclIsTypeOf (OpenElement))[Expression] will be converted to ExamElement.refImmediateComposite() [Expression]. When *d* is the source: updateBinding(RulesBySource(c), Bindings(RulesBySource(c)), b2), where in *b2* the name of the composition relation will be introduced in the path of the binding. For instance, OpenElement.question [Expression] must be converted to OpenElement.examElement.question [Expression].

Generalize Supertype (c∈ C, s∈ C, d∈ C)	deleteBinding(RulesBySource(c), Binding (RulesBySource(c), Metaproperties(s))).
InlineSubclass (c∈ C, d∈ C)	deleteRule(RulesBySource(c)) and moveBinding (RulesBySource(c), Bindings(RulesBySource(c)), RulesBySource(d)).
ReferenceTo Identifier (c∈ C, d∈ C, p∈ P)	(As a convention, the id will have the same name as the deleted reference) updateBinding(RulesBySource(c), Binding (RulesBySource(c), p), newBinding), where the *newBinding* will replace *reference* by *metaclass.id,* e.g. if metaclass *C* with a relation *p* to *D* is converted to *C* with a metaproperty referring to the new id in *D*, bindings *p* <- *D* (being *D* a reference to the generated element of type *D*) will be adapted to *p* <- *D.p.*
SplitReferenceBy Type (c∈ C, d∈ C, x∈ C, s∈ C, p∈ P, y∈ P, z∈ P)	deleteBinding(RulesBySource(d), p) and if *x* and *s* elements are created in the same rule:addBinding(RulesBySource(d), new_b).
PropertyMerge (p∈ P, z∈ P, x∈ P) if p, z, x ∈ SourceProperties	updateBinding(_, Binding(_, p), newBinding) and deleteBinding(_, Binding(_, z)), where newBinding will use *x* instead of *p* and *z*.

ClassMerge (c∈ C, d∈ C) if c, d ∈ SourceClasses	deleteRule(RulesBySource(c)) and deleteRule (RulesBySource(d)) and addRule(rule(_, concatClass(c, d), union(TargetsOfRule (RulesBySource(c)), TargetsOfRule(RulesBySource(d))), _ , union(Bindings (RulesBySource(c)), Bindings(RulesBySource(d)))). If there are filters in the rules: updateSource(_, c, concat(c, d)) and updateSource(_, d, concat(c, d)).
ClassMerge (c∈ C, d∈ C) if c, d ∈ TargetClasses	deleteTarget (RulesByTarget(c), c) and deleteTarget(RulesByTarget(d), d) and addTarget(RulesByTarget(c), concatClass(c, d)) and addTarget(RulesByTarget(d), concatClass(c, d)).
SplitClass (c∈ C, d∈ C)	deleteRule(RulesBySource(c)) and addRule(rule(_, d, TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding(RulesBySource(c), Metaproperties(d)))) and addRule(rule(_, SplitClassName(c), TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding (RulesBySource(c), Metaproperties(SplitClassName(c))))) .
PushMetaproperty (p∈ P)	(like move metaproperty)

4.5.4 The Case of the *removeProperty* Change

When a metaclass or a metaproperty is deleted, affected transformation elements have to be removed while keeping the transformation logic

L_1	L_2	L_1 AND L_2	L_1OR L_2	NOT L_1
R_T	L_2	L_2	R_T	R_F
R_F	L_2	R_F	L_2	R_T
R_T	R_F	R_F	R_T	-
R_T	R_T	R_T	R_T	-
R_F	R_F	R_F	R_F	-

Table 4.6: Truth table for removed elements.

coherent. Coherence refers to deleting only the strictly necessary parts to prevent negative consequences. For instance, two rules might exist with complementary filters. Those filters may refer to a property. If the deletion of this property leads to the removal of the whole filter, these two rules will no longer have a discriminating filter. Therefore, the impact of metamodel element deletions should be as restrictive as possible. This is specially pressing for rule filters. This subsection discusses a way to "surgically" remove "dead" parts of rule filters. Casuistic includes:

- *Expressions with string concatenation*: This is the easiest case, let be *style <- fontName + fontColor;* an expression with the concatenation of two string metaproperties, if one of them *(e.g. fontName)* is removed, then the expression is re-adapted to contain the rest of the metaproperties, i.e. the new expression is changed to *style <- fontColor.*

- *Expressions with creator operations of collections:* Collection types are *sets*, *ordered sets*, *bags*, and *sequences*. With expressions like *Set{London, Paris, Madrid} -> union(Set{birthCity, liveCity, workCity})*, after removing a metaproperty (e.g. *birthCity*), the new expression will keep the rest of the elements, i.e *Set{London, Paris, Madrid} -> union(Set{liveCity, workCity}).*

- *Expressions with other operations on collections:* There are other operations to work with collections, as *append(obj)*, *excluding(obj)*,

including(obj), *indexOf(obj)*, *insertAt(index, obj)*, and *prepend(obj)*. In this case, if the removed metaproperty is the parameter of the function, this part of the expression is removed. So, with an expression like *Set{London, Paris, Madrid} -> append(workCity)*, after removing the *workCity* metaproperty, the new expression will maintain the left hand of the expression, i.e *Set{London, Paris, Madrid}*.

- *Boolean expressions:* Since a removed metaproperty cannot be evaluated, that element in the expression must be considered as undefined. Moreover, before rewriting the expression with that undefined part, it is convenient to simplify the expression as much as possible, i.e. converting it into another equivalent expression, easier to deal with. Thus, equivalence rules of *Predicate Calculus* are applied to each boolean expression to get its equivalent CNF. Inspired by [CC04], table 4.6 is proposed as truth table which defines conversion rules for CNF expressions. In this table, R_T value will be interpreted as true, and R_F value as false. *L* represents a literal, which is an OCL expression that can be evaluated to a boolean value and does not include a boolean change.

  As an example, consider a metamodel with three metaproperties: *ErasmusGrant*, that says if the student has an Erasmus type grant; *speakEnglish*, that says if s/he has a good English level; and *enrolledLastYear*, that indicates if s/he is in his/her last undergraduate year. In the process of metamodel redesign, the designer could help giving some clue about the reason to take the decision of removing a metaproperty from the metamodel (*removal policy*). For example, if all students in the university had a very good level of English (because it is a new precondition for the enrollment), it could be considered as satisfied by default, and in case of removing the *speakEnglish* metaproperty, its value could be reinterpreted as *removed&satisfied-by-default* (R_T). On the other

hand, if the university had decided not to participate in the Erasmus Program, no student would have such grant, and in case of removing the *ErasmusGrant* metaproperty, its value could be reinterpreted as *removed&unsatisfied-by-default* (R_F).

If, in the previous example, there had been this expression *not ErasmusGrant or (speakEnglish and enrolledLastYear)*, and later the redesign process decided to remove the *speakEnglish* metaproperty, then, according to the truth table, the expression would be rewritten as *not ErasmusGrant or enrolledLastYear*; if the removed metaproperty had been *ErasmusGrant* with R_F policy, then, the new expression would have been *true.*

- *Expressions with loop operations:* In ATL, the syntax used to call an iterative expression is the following: *source -> operation_name (iterators | body)*. Among these operations, there are: *any(expr)*, *collect(expr)*, *exists(expr)*, *forAll(expr)*, *one(expr)*, *select(expr)*, and so on. For instance, in *self.items -> exists(i | i.question.size()>50),* if the removed metaproperty (e.g. *items*) takes part in the source, the whole expression is removed, but if the removed metaproperty takes part in the body, the rules for the boolean expressions must be applied.

- For more ambiguous cases, we resort to reporting the ambiguity and letting the designer decide. For instance, if the returned type of a helper is removed, the helper cannot be considered during binding, and a warning note is introduced. Or if the removal of a property makes the scope of two rules coincide then, the first one is commented.

Back to our second scenario (i.e. removal of *optional* from *ExamElement*), consider we have two rules whose filters refer to *optional*:

- *(value>5 and optional) or long*: Applying equivalences from table 4.6, the evolved filter becomes *(value > 5 or long*)

- *not ((value>5 and optional) or long)*: Using Morgan's laws, its CNF counterpart is: *(not value>5 or not optional) and not long*. Applying equivalences from table 4.6, the evolved filter results in *(not value>5 and not long)*.

In this way, "surgical removal" permits to limit the impact of deletion of properties in the associated rules.

4.5.5 The Case of *addClass* and *addProperty* Changes

```
1  rule MultipleChoice {
2    from xml : ExamXML!MCElement
3    to controller : AssistantMVC!MultipleChoiceController,
4      view : AssistantMVC!MultipleChoiceView (
5        controller <- controller,
6        fontName <- xml.attr1,
7        fontColor <- xml.attr2,
8        --fill right part
9        style <- xml.style )
10 }
11 --NEW RULE
12 rule ExerciseElement {
13   from s : ExamXML!ExerciseElement
14   to t : AssistantMVC!Exam (
15     --write the bindings
16     --target <- s.source)
17 }
```

Listing 4.3: Generated skeletons for the new *style* property (above) and the new *ExerciseElement* class (below).

Although additive evolution is considered *NBC*, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose, it is included in the co-evolution the option to generate partially new rule skeletons. The fifth scenario illustrates this situation: addition of the *ExerciseElement* metaclass, and addition of the *style* property to the *View* metaclass. The engineered co-evolution can be seen

at work in Figure 4.3: a rule is partially generated to tackle the addition of a new source metaclass while a new partial binding is proposed to address new properties. In the latter case, only a simple binding is generated (e.g. *target_metaproperty* <- *source_metaproperty*) which needs to be completed by the designer (in the example, *xml.style*).

4.6 Implementation

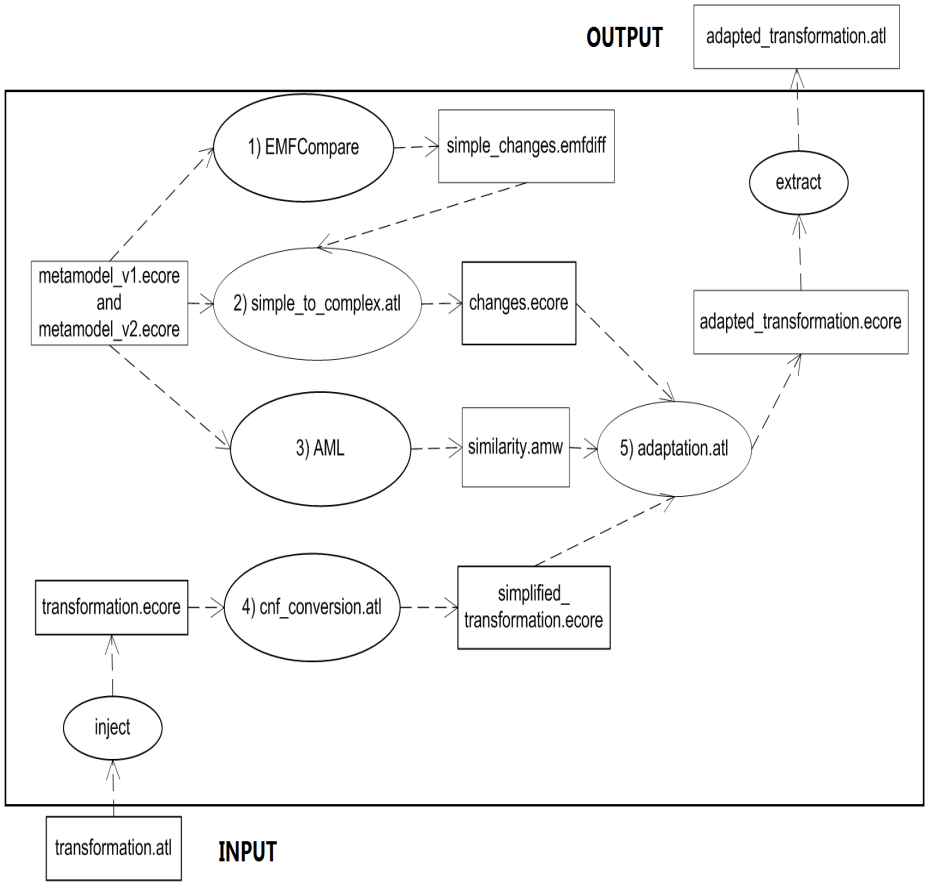

Figure 4.8: Architecture

The prototype is available[2] as a proof-of-concept of the feasibility of this approach for ATL rule co-evolution. Figure 4.8 depicts the main

[2]www.onekin.org/downloads/public/examination-assistant.rar

modules that mimic the co-evolution workflow introduced in Section 4.2. It takes an ATL file (.atl), two Ecore metamodels (.ecore) as input, and returns an ATL file that tackles the differences between the input Ecore models.

```
1  helper def: deleteRule_Splitclass (param : Sequence(ATL!
       MatchedRule)) : Sequence(ATL!MatchedRule) =
2    let elements : Sequence(String) = self.getSplittedClasses
3    in elements->iterate(p; y : Sequence(ATL!MatchedRule) =
         param |
4      if self.contains(p, param) then
5        self.deleteRule_Splitclass(y->excluding(param->at(
             self.index(p, param))))
6      else
7        y
8      endif);
9
10 rule Module_Splitclass {
11   from s : ATL!Module(
12     self.getUpdateAttributeRight_Splitclass.size()>0
13   to t : ATL!Module (
14     elements <- self.deleteRule_Splitclass(s.elements)
15     )
16   do{
17   t.elements <- t.elements->append(self.
         MatchedRule2MatchedRule_Splitclass(t.elements));
18   t.elements <- t.elements->append(self.
         MatchedRule2MatchedRule2_Splitclass(t.elements));}
19 }
20
21 lazy rule MatchedRule2MatchedRule_Splitclass {
22   from s : ATL!MatchedRule
23   to mr : ATL!MatchedRule ()
24     [...]
25   do{
26   for (iterator in self.simpleOutPatternElements){
27   op_i_c2.elements <- op_i_c2.elements->append(thisModule.
         SOPE2SOPE_Splitclass(op_i_c2.elements));
```

```
28    self.index_Splitclass <- self.index_Splitclass + 1;}}
29  }
30
31  lazy rule SOPE2SOPE_Splitclass {
32    from s : ATL!SimpleOutPatternElement
33    to ope_i_c2 : ATL!SimpleOutPatternElement()
34    do{
35    self.indexBinding_Splitclass <- 1;
36    for (iterator in self.simpleOutPatternElement.at(self.
        index).bindings){
37      if (self.simpleOutPatternElements.at(self.
          index_Splitclass).bindings.at
38        (self.indexBinding_Splitclass).value.oclIsTypeOf(ATL!
            VariableExp)){
39        ope_i_c2.binding <- ope_i_c2.bindings->append(self.
            B2B_Splitclass(ope_i_c2.bindings));
40      }else{
41        if (self.simpleOutPatternElements.at(self.
            index_Splitclass).bindings.at
42          (self.indexBinding_Splitclass).value.oclIsTypeOf(
              ATL!StringExp)){
43          ope_i_c2.bindings <- ope_i_c2.bindings->
44          append(self.B2BString_Splitclass(ope_i_c2.bindings)
              );}
45      self.indexBinding_Splitclass <- self.
          indexBinding_Splitclass + 1;}}
46  }
47
48  lazy rule B2B_Splitclass {
49    from s : ATL!Binding
50    to b : ATL!Binding (
51      [...]
52  }
```

Listing 4.4: HOT rules to cope with class split. HOTs' input and output models conform to the ATL metamodel.

The main effort was devoted to the adaptation module. Implementation wise, this module also represents the main innovative approach since

transformation co-evolution is achieved using HOT transformations. Along the MDE *motto*: "*everything is a model"* [BÓ4], transformations are models that conform to their own metamodel (i.e., the transformation language). Being models, *(Higher Order)* transformations can be used to map the original transformation model into a co-evolved transformation model that caters for the metamodel changes. Figure 4.4 outlines one such HOT transformation that tackles the *splitClass* case. The pattern includes: a "main" rule, some lazy rules that are called from it to create elements, and some helpers to modularize the functionality. In this specific case, *Module_Splitclass* is the "main" rule (line 9), which will be executed when there is any change of type *Splitclass*. In the *to part* of the rule, the *deleteRule_Splitclass* helper is called (line 13), which causes the deletion of the rule referring to the deleted metaclass. Then, the imperative part of the rule (*do*) creates two new rules: *MatchedRule2MatchedRule_Splitclass* and *MatchedRule2MatchedRule2 _Splitclass* (lines 16 and 17). These rules in turn refer to *SOPE2SOPE _Splitclass* rule (line 24), which creates a *SimpleOutPatternElement* for the new generated rules. Finally, this rule invokes *B2B_Splitclass* to generate the bindings (lines 34 and 39).

4.7 Related Work

Although co-evolution of models after metamodel evolution has been widely studied [CREP08, HVW11, RKPP10], transformations have raised less attention. A lot of research has been carried out in the model co-evolution area and some proposals have been done to semi-automatically adapt models to metamodel evolution. Three main strategies have been used [Her11]: *(i)* manual specification: these approaches provide transformation languages to manually specify the migration (e.g. [RKPP10]); *(ii)* matching approaches: they intend to automatically derive a migration from the matching between two metamodel versions (e.g. [CREP08]); and *(iii)* co-evolution based on operators: they record the coupled operations which are used to evolve the metamodel and which

also encapsulate a model migration (e.g. [HVW11]). Following this classification, the approach presented in this chapter would be in the second type, as we do not know changes in advance or make them in any specific tool. But on the other hand, we rely the set of complex changes in a taxonomy of operators based on the third type ([HVW11]). This approach is similar, as each change has an associated co-evolution, but the difference is that we do not create explicitly the operators, as they are automatically derived. In some cases changes in metamodels do not affect transformations, as studied in [SJ07], where authors conclude that the addition of new classes and broadening of multiplicity constraints do not break the subtyping relationship between metamodel versions. But often changes do have an impact on transformations. To the best of my knowledge, two authors ([LBNK10] and [SDP+10]) have dealt with transformation co-evolution. The first case is limited to graph-based languages, considering simple changes and considering subtractive changes only as coarse-grained removals (i.e., rule level deletions). In contrast, the presented approach is focused on rule-based declarative languages, deleting as little as possible, and considering complex changes. In [SDP+10] authors explain a fundamental idea, e.g., the convenience of using operators in the co-evolution of transformations. Compared to this chapter's approach, the contribution would be an automatic conversion from simple to complex changes, minimum deletion and an implementation of co-evolutions in ATL. First issue of the approach, the conversion of simple to complex changes is treated in [GJCB09b] and [VWV12]. The former is based on a DSL for expressing model matching and the later uses a sequence of operator instances as evolution trace, and they allow to make changes over changes.

The co-evolution process only guarantees that the transformation is syntactically correct, and if other correctness properties need to be checked, other complementing works will have to be considered, as analysis and simulation [ABK07], testing [KAER06] or metamodel coverage [WKC06]. In the case where co-evolution is done manually,

coverage analysis can be used to determine whether the changes to a metamodel affect the transformation [vAvdB11].

4.8 Conclusions

It has been addressed how metamodel evolution can be semi-automatically propagated to the transformation counterpart. The process flow includes: (1) detecting simple changes from differences between the original metamodel and the evolved metamodel, (2) deriving complex changes from simple changes, (3) translating boolean expressions to the CNF form, (4) if available, capitalize on model similarity, and finally, (5) map the original transformation into an evolved transformation that (partially) tackles the evolved metamodel. The approach is realized for EMOF/ Ecore-based metamodels, and ATL transformations. The approach relieves domain experts from handling routine cases so that they can now focus on the more demanding scenarios (e.g. additive evolution). The use of high-level transformations implies the existence of a transformation metamodel. So far this is available for main transformation languages such as ATL or RubyTL.

Chapter 5

An Adapter-Based Approach to Co-Evolve Generated SQL in Model-to-Text Transformations

5.1 Introduction

The changing nature of DB schemas has been a constant concern since the inception of DBs. Software consuming data is dependent upon the structures keeping this data, i.e. the DB schema. Such software might refer to relational views [CMDZ13], data mappings (i.e. describing how data instances of one schema correspond to data instances of another) [VMP04] or application code [CH05]. But, what if this software is not directly coded but generated? *Forward Engineering* advocates for code to be generated dynamically through model-to-text transformations that target a specific platform. In this setting, platform evolution can leave the transformation, and hence the generated code, outdated. This issue is exacerbated by the *perpetual beta* phenomenon in Web 2.0 platforms where continuous delta releases are a common practice.

The issue of keeping the application and the DB schema in sync is turned into one of maintaining the consistency between the code

generators (i.e. the transformation) and the DB schema. Despite the increasing importance of Forward Engineering, this issue has been mostly overlooked. This might be due to understanding that traditional co-evolution techniques can be re-used for transformations as well. After all, transformations are just another kind of applications. However, existing strategies for application co-evolution assume that either the generated SQL script is static or the trace for DB changes is known (and hence, can later be replicated in the application) [CH05]. However, these premises might not hold in this scenario. Rationales follow:

- transformations do not *specify* but *construct* SQL scripts. The SQL script is dynamically generated once references to the input model are resolved. This specificity of transformations makes it necessary to do the adaptation dynamically after the transformation engine has resolved the references but before it prints the result to the output file. As a result, solutions proposed for "static" scenarios are hardly applicable in this context.
- the trace for DB changes might be unknown. The DB schema and the transformation might belong to different organizations (i.e. there exist an *external* dependency from the transformation to the schema). This rules out the possibility of tracking schema upgrades to be later replicated into the transformation. External dependencies are increasingly common with the advent of the Web 2.0, and the promotion of open APIs and open-source platforms such as Content Management Systems (e.g. *Alfresco*), Wiki engines (e.g. *MediaWiki*) or Blog engines (e.g. *WordPress*). Here, the application (i.e. the portal, the wiki or the blog) is developed on top of a DB schema that is provided by a third party (e.g. the *Wikimedia* foundation).

The aforementioned scenario was faced when implementing *WikiWhirl* [PD12], a Domain-Specific Language (DSL) built on top of *MediaWiki*. *WikiWhirl* is interpreted, i.e. a *WikiWhirl* model (an expression described

along the *WikiWhirl* syntax) delivers an SQL script that is enacted. The matter is that this *SQL* script goes along the *MediaWiki* DB schema. If this schema changes, the script might break apart. Since *MediaWiki* is part of the *Wikimedia Foundation*, we do not have control upon what and when *MediaWiki* releases are delivered. And release frequency can be large, which introduces a heavy maintenance burden upon *WikiWhirl*.

To tackle the mentioned problem, the transformation process is split in two parts: the stable part is coded as a transformation, whereas the unstable side is isolated through an adapter that is implicitly called by the transformation at generation time. In this way, platform upgrades impact the adapter but leave the transformation untouched. That is, data manipulation requests (i.e. insert, delete, update, select) are re-directed to the adapter during the transformation. The adapter outputs the code according to the latest schema release. In this way, the main source of instability (i.e. schema upgrades) is isolated in the adapter.

The chapter starts by framing the contribution within the literature in DB schema evolution (Section 5.2). Next, it is introduced *WikiWhirl* as a motivating scenario (Section 5.3), and continue comparing different solution alternatives (Section 5.4). Then, it is described the two main stages: change detection (Section 5.6) and change propagation (Section 5.7). It follows a cost-effective evaluation in Section 5.8. Later, it is explained how the changes done in the code are dumped into the transformation. Conclusions end the chapter.

5.2 Adapters as Artifact Synchronizers

Synchronization is the process of enforcing consistency among a set of artifacts, and *synchronizers* are procedures that automate—fully or in part—the synchronization process. Synchronization deserves attention, since 40% of MDE developers complain about the time they spend on synchronization tasks [HWRK11]. In chapter 3, an overview of the synchronization design space was provided. This section formalizes and

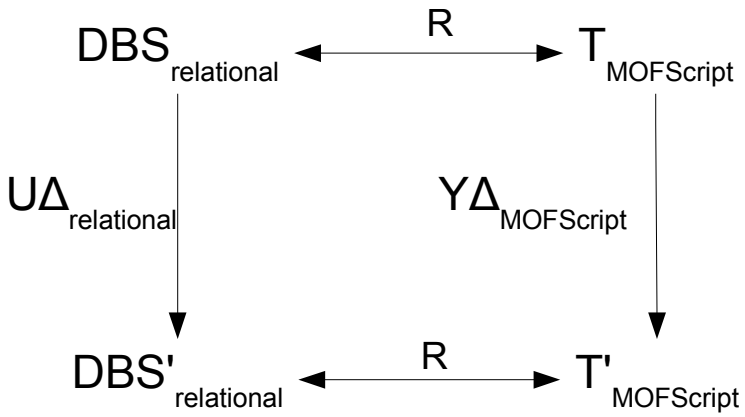

Figure 5.1: Artifacts involved in unidirectional co-evolution. *R* denotes the consistency relationship. (Adapted from [AC07]).

frames this chapter within this space.

When an artifact is modified, related artifacts need to be updated in order to re-establish existing consistency relations. Let $DBS_{relational}$ and T_{M2T} be two artifacts of type relational and *M2T* (Model to Text), respectively (i.e. the set of database schema compliant with the relational model, and the set of *M2T* transformations). We say that artifacts $DBS_{relational}$ and T_{M2T} are consistent or synchronized with respect to the consistent relation R iff T_{M2T} refers to existing tables/columns defined in DBS where mandatory columns are initialized. If tables are enlarged, reduced, deleted or renamed or columns are renamed, then, this consistency is broken.

Unlike bidirectional synchronization, a unidirectional synchronization only computes the target artifact (e.g. T_{M2T}) from the source (e.g. $DBS_{relational}$), but not vice versa. This unidirectional synchronization is said to be "to-one" if the computation outputs a single target artifact. This computation can be realized in three different ways. First, using artifact translation, i.e. an operator that translates an entire source artifact into a consistent target artifact. The second option uses heterogeneous artifact comparison, an operator that directly compares two artifacts of different

types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact. The third variant uses update translation, i.e. an operator that translates an update to the source artifact into a consistent update of the target artifact. Our approach falls within the last category: unidirectional synchronization in to-one direction using update translation. Figure 5.1 depicts the main artifacts involved:

- $DBS_{relational}$ is the original DB schema;
- T_{M2T} is the original legacy application (i.e., the version of the transformation that co-existed with the original DB);
- $DBS'_{relational}$ is the new DB schema;
- T'_{M2T} is the new target artifact; $U\Delta_{relation}$ is the update applied to the original source[1];
- $Y\Delta_{M2T}$ is the target update resulting from the coevolution.
- $U\Delta_{relation}$ can be obtained by recording the changes upon $DBS_{relational}$ while the user edits it [CH05]. However, this is not always possible since the schema belongs to a third party.

Alternatively, an **update** can be computed using a homogeneous artifact comparison operator, which takes an original version of an artifact and its new version, and returns an update connecting the two. Now, it can be obtained transformation update (i.e. $Y\Delta_{M2T}$) *out of* $U\Delta_{relation}$. It is proposed that transformations are engineered for evolution. Therefore, a transformation is conceived as a pair (S, N) where *S* denotes the stable part, and *N* stands for the no-stable part. The no-stable part is supported through an adapter. Therefore, $Y\Delta_{M2T}$ denotes the update to be conducted in the adapter. Therefore, it is presented an architecture to compute adapter updates (i.e. $Y\Delta_{M2T}$) out of differences between DB schema

[1] An **update** is a function that takes an artifact as input, and returns the updated artifact as output, e.g. $DBS'_{relational} = U\Delta_{relational}(DBS_{relational})$.

models (i.e. $U\Delta_{relation}$). The idea is for the adapter to be domain-agnostic, and hence, reusable in other domains. It is envisaged transformation adapters to play a role similar to DBMS drivers. A DBMS driver shelters applications from the heterogeneity of DBMS. The driver translates the application's data queries into commands that the DBMS understands. Likewise, a transformation adapter seeks to isolate the transformation from changes in the DB schema. As long as these changes are domain agnostic (e.g. the way to face attribute rename is domain independent) then, the adapter can be re-used by the community. The solution is available in *www.onekin.org/downloads/public/Batch_MofscriptAdaptation.rar*. There are also three explanation screencasts in *http://onekin.org/downloads/public/screencasts/MOFScript/*.

5.3 Case Study

```
1  texttransformation FreeMind2MediaWiki (in ww:"www.onekin.
       org/wikiwhirl",
2      in diff: "http://www.eclipse.org/emf/compare/diff/1.1",
3      in Ecore: "http://www.eclipse.org/emf/2002/Ecore") {
4      var timestamp : String = "('%Y%m%d%k%i%s')"
5      var categoryTitle :  String = wikiRes2.title
6      var userId: String = "1"
7                [...]
8       ww.Categorize:categorize_sql(){
9           println("UPDATE page set page_touched = " +
                timestamp +
10        " where page_namespace = 14 and page_title = '" +
              categoryTitle + "';")
11        println("INSERT into recentchanges (rc_timestamp,
              rc_cur_time, rc_user,
12        rc_user_text, rc_namespace, rc_title, " + "
              rc_comment, rc_new, rc_cur_id,
13        rc_this_oldid, rc_last_oldid, rc_type, rc_old_len,
              rc_new_len, rc_deleted)
14        VALUES (" + timestamp + ", " + timestamp + ", " +
```

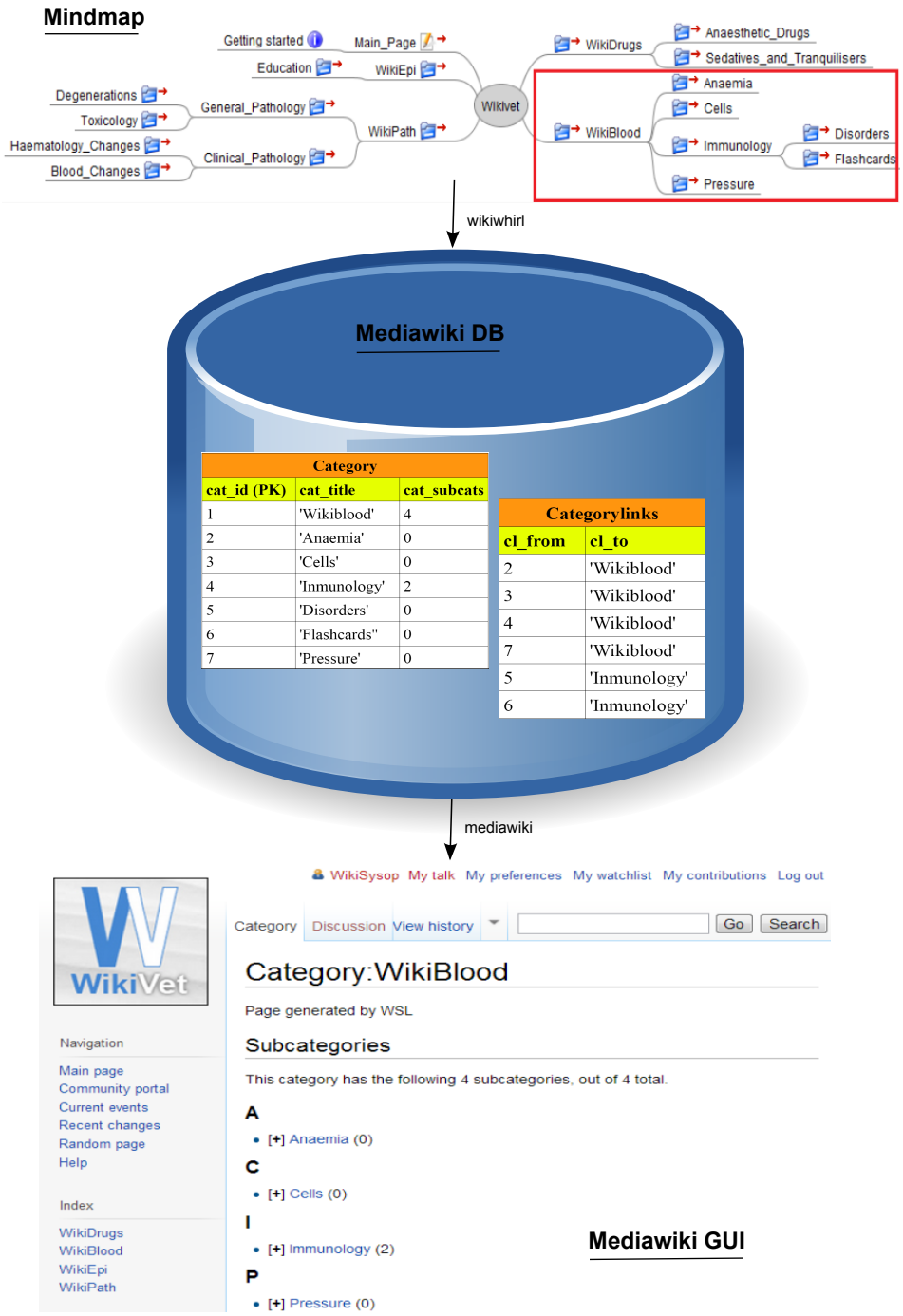

Figure 5.2: WikiWhirl overview

```
            userId + ", '" + userName +
15      "', " + namespace + ", '" + pageTitle + "', '" +
            comment + "', 0, pageId, " +
16      lastRevisionId + ", " + pageRevisionId + ", 0, " +
            pageLen + ", " + pageLen +
17      " - " + categoryTitle.size() + " - " + "[[Categoy:]]
            ".size()  + ", 0);")
18  }       [...]
```

Listing 5.1: A *WikiWhirl* transformation snippet. The transformation constructs the SQL statements dynamically from the input model; i.e. the *WikiWhirl* expression (e.g. *categortyTitle* in line 10; *pageTitle* in line 15; etc)

This section outlines the *WikiWhirl* project [PDA13] and the challenges posed by its external dependency with the *MediaWiki* DB schema. Wikis are main exponents of collaborative editing where users join forces towards a common goal (e.g. writing an article about a topic). It comes as no surprise that wikis promote an article-based view as opposed to a holistic view of the wiki content. As a result, APIs and GUIs of wiki engines favor operations upon single articles (e.g. editing and discussing the article's content) while overlooking operations on the wiki as a whole (e.g. rendering the wiki's structure or acting upon this structure by splitting, merging or re-categorizing articles). To amend this, *WikiWhirl* abstracts wiki structure in terms of mindmaps, where refactoring operations (*WikiWhirl* expressions) are expressed as reshapes of mindmaps. Since wikis end up being supported as DBs, *WikiWhirl* expressions are transformed into SQL scripts. This process is summarized in Figure 5.2. Figure 5.1 shows a snippet of a *WikiWhirl* transformation: a sequence of SQL statements with interspersed dynamic parts that query the input model (i.e. the *WikiWhirl* expression). These statements built upon the DB schema of *MediaWiki*, and in so doing, create an external dependency of *WikiWhirl* w.r.t. *MediaWiki*. *MediaWiki* is a wiki

Approach	Involvement	Skills	Infrastructure skills
manually changing the generated code	High	SQL	default
manually changing the transformation	High	SQL, MOFScript	default
automatically changing the transformation	Low	SQL, ATL	ATL Injectors, ATL metamodel, SQL schema Injectors, Model differentiation
automatically *adapting* the transformation	Low	SQL	SQL schema Injectors, Model differentiation

Table 5.1: Alternatives to manage platform evolution.

engine, currently used by almost 40,000 wikis[2]. In a 4½ year period, the *MediaWiki* DB had 171 schema versions [CTMZ08]. According to [CTMZ08], the number of tables has increased from 17 to 34 (100% increase), and the number of columns from 100 to 242 (142%). Rationales include performance improvements or the addition of new features. These changes can eventually break those applications that depend upon the *MediaWiki* DB schema. Figure 5.2 illustrates the problem: in the upper part of the figure there is a mindmap that represents a wiki. That mindmap will be manipulated by the user in order to refactor the wiki; and finally that changes will be propagated to the actual wiki, in the bottom of the figure. This begs the question of how to make *WikiWhirl* co-evolve with the *MediaWiki* upgrades. Next section compares four options in terms of the involvement (i.e. time and focus) and the required technical skills.

5.4 Solution Alternatives

Option 1: Manually Changing the Generated Code

The designer detects that the new release impacts the generated code,

[2]http://s23.org/wikistats/

and manually updates this code. This approach is discouraged in Forward Engineering outside the protected areas, since subsequent regenerations can override the manual changes. On the upside, this approach acts directly on the generated code, so only SQL skills are required to accomplish the change.

Option 2: Manually Changing the Transformation

For sporadic and small transformations, this might be the most realistic option. However, frequent platform releases and/or SQL-intensive transformations make this manual approach too cumbersome and error-prone to be conducted in a regular basis. The user needs to know both the platform language (e.g. SQL) and the transformation language (e.g. *MOFScript*). No additional infrastructure is introduced.

Option 3: Automatically Changing the Transformation

The idea is to inject the transformation into a model and next, use a *Higher-Order Transformation* (HOT) [TJF+09] to upgrade it. HOTs are used to cater for transformation variability in Software Product Lines [OH07]. Variability is sought to generate code for different platforms, different QoS requirements, language variations, or target frameworks. The approach is to define "aspects" (i.e. variability realizations) as higher-order transformations, i.e. a transformation whose subject matter is another transformation. Using aspect terminology, the *pointcuts* denote places in a transformation that may be modified by the high-order transformation (HOT). *Advices* modify base transformations through additional transformation code to be inserted before, after, or to replace existing code. Likewise, we could rephrase schema co-evolution as a "variability-in-time" issue, and use HOTs to isolate schema upgrades. Unfortunately, the use of HOTs requires of additional infrastructure: (1) a metamodel for the transformation language at hand, and (2) the appropriate injector/extractor to map from the *MOFScript* code to the *MOFScript* model, and vice versa. The availability of these tools is not always guarantee. For instance, *MOFScript* has both an injector and an extractor. However, *Acceleo* lacks the extractor. Another drawback is generality. It

could be possible to develop a HOT for the specific case of *WikiWhirl*. But we aim at the solution to be domain-agnostic, i.e. applicable to no matter the domain meta-model. Unfortunately, HOTs find difficulties in resolving references to the base transformation's input model where its metamodel is unknown at compile time (i.e. where the HOTs is enacted). Moreover, this approach requires additional infrastructure: (1) SQL schema injectors that obtain a model out of the textual description of the DB schema and (2), a model differentiation tool that spots the differences among two schema models. This infrastructure is domain-independent.

Option 4: Automatically Adapting the Code Generated by the Transformation

The user does not need to look at the transformation (which is kept unchanged) but at the generated code. SQL skills are sufficient. At runtime, the transformation invokes the adapter instead of directly generating the DB code (i.e. the SQL script). This brings two important advantages. First, and unlike the HOT option, the issue of resolving references to the base transformation's input model, does not exist, as references are resolved at runtime by the transformation itself. Second, this solution does not require the model representation of the transformation (i.e. injectors for the transformation are not needed). On the other side, this approach requires, as option 3, SQL schema injectors and a model differentiation tool.

5.5 Process Overview

Figure 5.3 outlines the different steps of the proposal. First, DB schemas (i.e. *New schema*, *Old Schema*) are injected as *Ecore* artifacts (step 1); next, the schema difference is computed (i.e. *Difference model*) (step 2); finally, this schema difference feeds the adapter used by the transformation (i.e. *MOFScript program*). *MOFScript* is a template-based code generator that uses *print* statements to generate code and language instructions to retrieve model elements. The approach mainly consists of replacing the

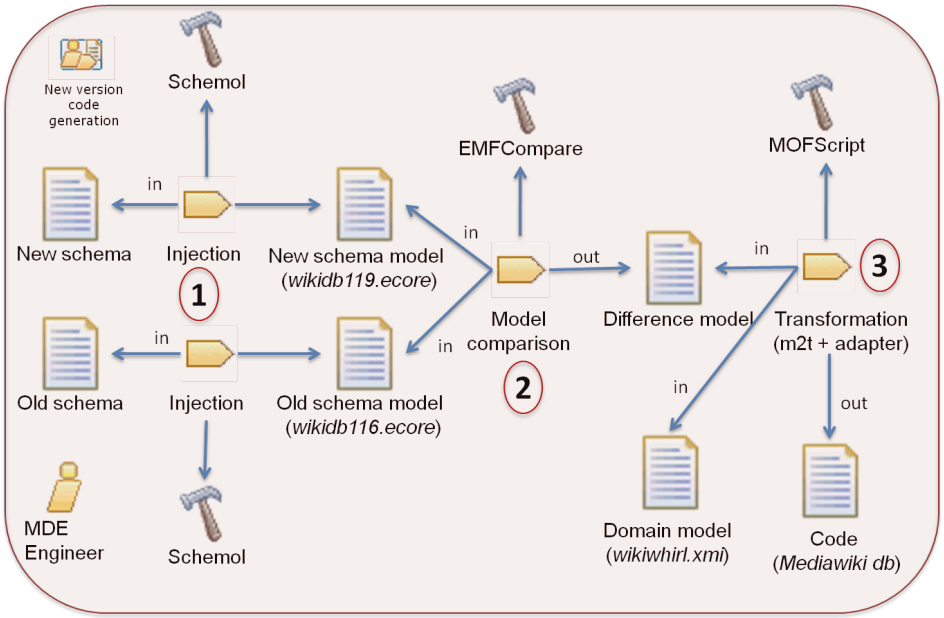

Figure 5.3: A generic co-evolution process, exemplified for the WikiWhirl case study.

print statements with invocations to the adapter (e.g. *printSQL*). On the invocation, the adapter checks whether the <*SQL statement*> acts upon a table that is being subject to change. If so, the adapter returns a piece of SQL code compliant with the new DB schema. Section 5.6 and Section 5.7 address change detection (steps 1 and 2) and change propagation (step 3), respectively.

5.6 Change Detection

Upgrades on the MediaWiki's DB schema are well documented[3]. Developers can directly access this documentation to spot the changes. Gearing towards automatization, these changes can also be ascertained by installing the new release, and comparing the old DB schema and the new DB schema (see Figure 5.4). The process starts by a notification of a new

[3]http://www.mediawiki.org/wiki/Manual:Database_layout

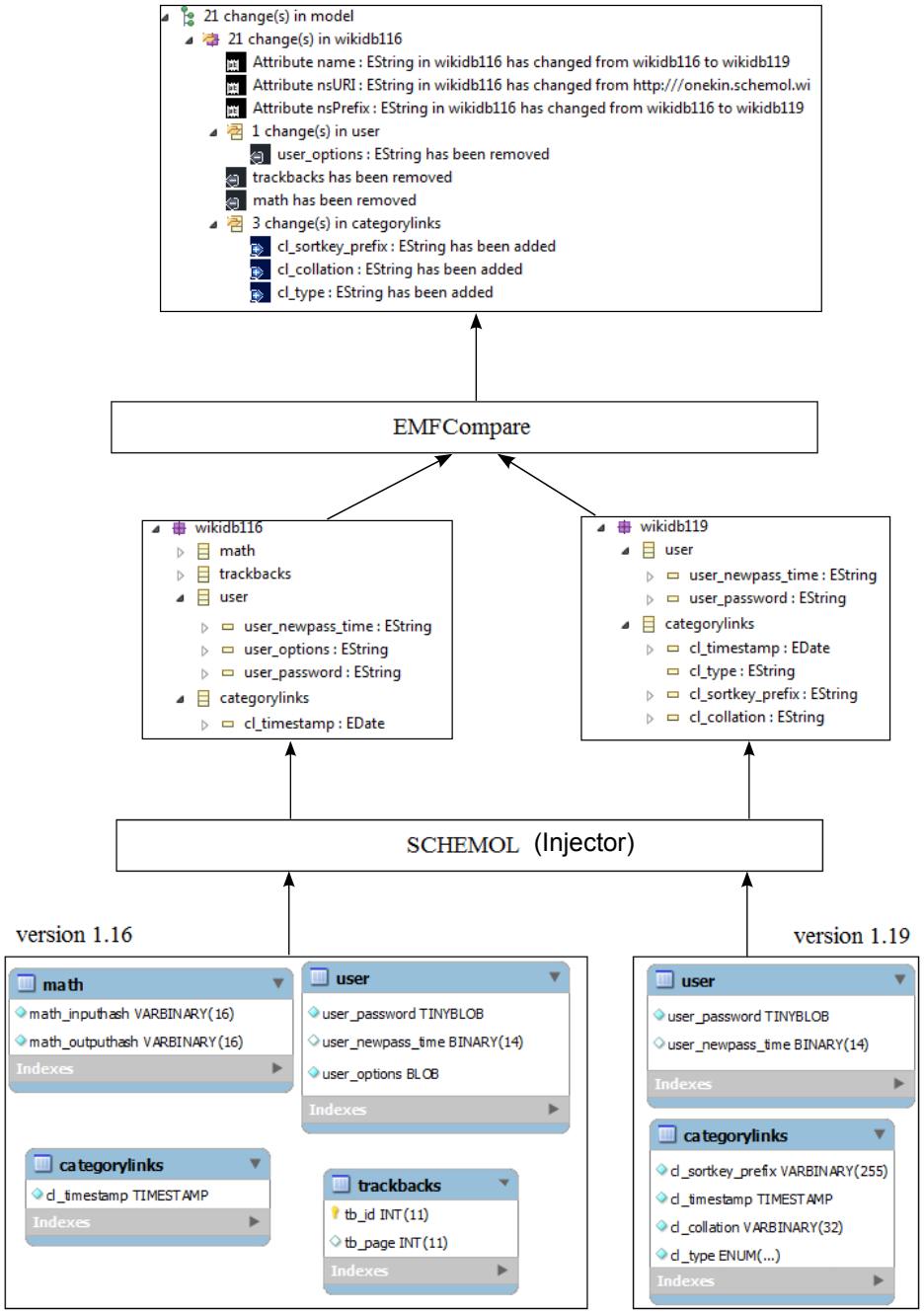

Figure 5.4: Injection: from catalog tuples (those keeping the DB schema) to the Difference model.

MediaWiki release (e.g. version 1.19). The developer obtains the model for the new schema (*wikidb119*) as well as the model of the schema used in the current release of *WikiWhirl* (*wikidb116*) using some schema injector (e.g. Schemol). Next, schema differences are computed as model differences (e.g. using *EMFCompare).* The output is the *Difference* model.

The *Difference* model is described as a sequence of DB operators. Curino et al. proved that a set of eleven *Schema Modification Operators (SMO)* can completely describe a complex schema evolution scenario. Table 5.2 indicates the frequency of these change for the *MediaWiki* case, elaborated from [CTMZ08]. Fortunately, most frequent changes (e.g. *'create table', 'add column', 'drop column'* or *'rename column*') can be identified from schema differences. Complex changes (e.g. *'distribute table'* or *'merge table*') cannot be automatically detected and therefore are not included in the table. This kind of changes tend to be scarce. For *MediaWiki*, *'distribute table'* never occurred while *'merge table'* accounts for 1,5% of the total changes.

5.7 Change Propagation

Schema changes need to be propagated to the generated code through the transformation. The transformation delegates to the adapter how the SQL command ends up being supported in the current DB schema. That is, *MOFScript's print* is turned into the adapter's *printSQL* (e.g. *"printSQL (<SQL statement>)"*). On invocation, the adapter checks whether the SQL statement acts upon a table that is subject to change (i.e. appears in the *Difference* model). If so, the adapter proposes an adaptation action to restore the consistency. This adaptation action depends on the kind of change: NBC, BRC or BUC. Based on this classification, different contingency actions are undertaken: no action for NBC, automatic co-evolution for BRC, and assisting the user for BUC. Table 5.2 describes this typology for DB changes, the usage percentage of each change for *MediaWiki* [CTMZ08], and the adaptation counterpart.

SMO	% of usage	Change type	Adaptation
Create table	8.9	NBC	New comment in the transformation on the existence of this table in the new version
Drop table	3.3	BRC	Delete statement associated to the table
Rename table	1.1	BRC	Update name
Copy table	2.2	NBC	(None)
Add column	38.7	NBC/ BRC	For *insert* statements: if the attribute is *Not Null*, add the new column in the statement with a default value (from the DB if there is available or according to the type if there is not)[4]
Drop column	26.4	BRC	Delete the column and value in the statement
Rename column	16	BRC	Update name
Copy column	0.4	BRC	Like *add column* case
Move column	1.5	BRC	Like *drop column* + *add column* cases

Table 5.2: *Schema Modification Operators* and their adaptation action counterparts.

```
1 printSQL(statement: String){
2     [...]
3      var tableName : String = java("org.gibello.zql.
          ZqlParser",
4      "getTableName", statement , CLASSPATH );
5      diff.objectsOfType(diff.RemoveModelElement)->forEach
6      (rme:diff.RemoveModelElement | rme.rightParent.name=
          tableName){
7          var paramsRemoveColumn:List;
8          paramsRemoveColumn.add(statement);
9          paramsRemoveColumn.add(rme.rightParent.name);
```

```
10          paramsRemoveColumn.add(rme.leftElement.name);
11          println("#"+statement);
12          println(java("org.gibello.zql.ZqlParser", "
                removeColumn",
13          paramsRemoveColumn, CLASSPATH) + ";");
14      }
15    [...]
16  }
```

Listing 5.2: The adapter. A *printSQL* function that handles the *remove column* case.

Implementation wise, the adapter has two inputs: the *Difference* model and the model for the new schema (to obtain the full description of new attributes, if applicable). The ZQL[5] open-source SQL parser was used to parse SQL statements to Java structures. This parser was extended to account for adaptation functions to modify the statements (e.g. *removeColumn*) and support functions (e.g. *getTableName*). Figure 5.4 provides a glimpse of the adapter for the case "*remove column*". It starts by iterating over the changes reported in the *Difference* model (line 5). Next, it checks (line 6) that the deleted column's table corresponds with the table name of the statement (retrieved in lines 3-4). Then, all, the statement, the table name and the removed column are added to a list of parameters (lines 7-10). Finally, the adapter outputs an SQL statement without the removed column, using a function with the list of parameters that modifies the expression (lines 12-13). The adaptation process is enacted for each *printSQL* statement, regardless of whether the very same statement has been previously processed or not. Though this penalizes efficiency, the frequency and the time at which this process is conducted make efficiency a minor concern.

```
1  //VERSION 1.16
2     INSERT INTO categorylinks (cl_from, cl_to, cl_sortkey,
          cl_timestamp)  VALUES
3       (@pageId, 'Softwareproject', 'House_Testing',
```

[5]http://zql.sourceforge.net/

```
4       (DATE_FORMAT(CURRENT_TIMESTAMP(), '%Y%m%d%k%i%s'));
5     INSERT INTO trackbacks (tb_name, tb_title, tb_url, tb_ex
          , tb_id, tb_page)
6       VALUES ('trackback1', 'title', 'http://blog/post', '',
            '', '');
7     INSERT INTO user (user_id, user_name, user_real_name,
          user_password,
8       user_newpassword, user_newpass_time, user_email,
            user_options,
9       user_touched, user_token, user_email_token_expires,
            user_registration,
10      user_editcount) VALUES ('1', 'Jokin', 'Jokin Garcia',
            'c7c1105fac', '', NULL,
11    'jokin.garcia@ehu.es', 'quickbar=1', '20110902', '
          d863a16e41', '', '20070718', '1360');
```

Listing 5.3: MediaWiki 1.16 generated script

```
1   //VERSION 1.19
2     //WARNING: Added columns cl_type, cl_sortkey_prefix and
          cl_collation
3     INSERT INTO categorylinks (cl_from, cl_to, cl_sortkey,
          cl_timestamp, cl_type,
4       cl_sortkey_prefix,cl_collation) VALUES (@pageId, '
            Softwareproject', 'House_Testing',
5       (DATE_FORMAT(CURRENT_TIMESTAMP(), '%Y%m%d%k%i%s'), '
            page', '', '0');
6     //WARNING: Deleted table trackbacks
7     //INSERT INTO trackbacks (tb_name, tb_title, tb_url,
          tb_ex, tb_id, tb_page)
8     //VALUES ('trackback1', 'title', 'http://blog/post', '',
            '', '');
9     //WARNING: Deleted column user_options in table user
10    //INSERT INTO user (user_id, user_name, user_real_name,
          user_password,
11    //user_newpassword, user_newpass_time, user_email,
          user_options,
12    //user_touched, user_token, user_email_token_expires,
          user_registration,
```

```
13  //user_editcount) VALUES ('1', 'Jokin', 'Jokin Garcia',
        'c7c1105fac', '', NULL
14  //'jokin.garcia@ehu.es', 'quickbar=1', '20110902', '
        d863a16e41', '', '20070718', '1360');
15  INSERT INTO user (user_id, user_name, user_real_name,
        user_password,
16    user_newpassword, user_newpass_time, user_email,
          user_touched,
17    user_token, user_email_token_expires,
          user_registration, user_editcount)
18    VALUES ('1', 'Jokin', 'Jokin Garcia', 'c7c1105fac', ''
          , NULL,
19    'jokin.garcia@ehu.es', '20110902', 'd863a16e41', '', '
          20070718', '1360');
```

Listing 5.4: MediaWiki 1.19 generated script. Since the MOFScript code keeps constant; differences are due to the adapter. The adapter also intermingles comments to ease user inspection.

Back to our sample case, the SQL script in Listing 5.4 is the result of enacting the generation process with *wikidiff_v16v19* as the *Difference* model. Once references to the variables and the model elements have been resolved, MOFScript's *printSQL* statements invoke the adapter. The adapter checks whether either the tables or the attributes of the *printSQL* statement are affected by the upgrade (as reflected in the *Difference* model) and applies the appropriate adaptation (see table Table 5.2). Specifically, the *Difference* model *wikidiff_v16v19* reports:

1. the introduction of three new attributes in the *categorylinks* table, namely, *cl_type*, *cl_sortkey_prefix* and *cl_collation*. Accordingly, the adapter generates *SQL* insert/update statements where new attributes which are 'Not Null' are initialized with their default values (lines 1-4 below);

2. the deletion of tables *math* and *trackback*. This causes the affected *printSQL* statements to output nothing (i.e. the old output is left as a comment) (lines 5-7 below);

Version	#Add column	#Impacts on WikiWhirl	#Drop column	#Impacts on WikiWhirl
1.17	1	3	0	0
1.18	0	0	1	1
1.19	2	11	1	0

Table 5.3: Co-evolving *WikiWhirl* from *MediaWiki* 1.16 to *MediaWiki* 1.19.

3. the deletion of attribute *user_options* in the *user* table. Consequently, the affected *printSQL* statements, output the SQL but removing the affected attributes (lines 14-18 below). In addition, a comment is introduced to note this fact (lines 8-13 below).

There is a video available that shows how an adaptation works [6].

5.8 Assessment

The net value of an adapter is given by the cost savings that occur to accommodate each DB release minus the development cost of the adapter. These savings are expected from *Breaking and Resolvable Changes (BRC)* since (1) they are amenable to be automated, and (2) they account for the majority of the change. Next paragraphs provide some figures for BRCs, comparing the manual vs the adapter-based approach. To this end, it has been conducted an assessment on the cost of migrating a *WikiWhirl*'s *MOFScript* transformation from version 1.16 to version 1.17 of the *MediaWiki* DB schema. Table 5.3 provides some figures of the schema changes and their impact. The effort is estimated in terms of the number of *MOFScript* instructions affected. The experiment was conducted by 8 PhD students who were familiarized with SQL, *MOFScript*

[6] http://onekin.org/downloads/public/screencasts/Adaptation.htm

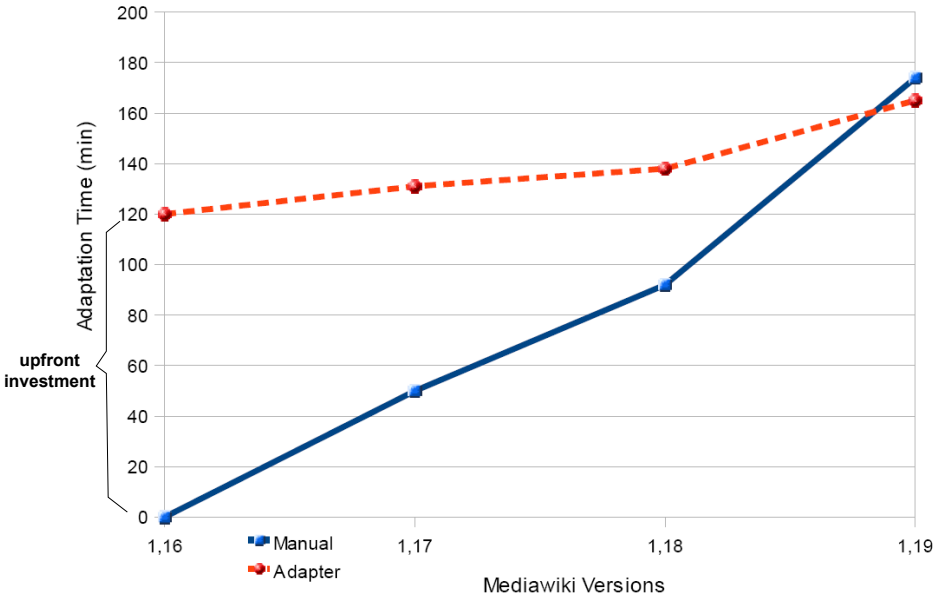

Figure 5.5: Accumulative costs of keeping *WikiWhirl* and *MediaWiki* in sync. Comparison of the manual (continuous line) and the assisted approach (dotted line).

and *Ant* [7] (4 for the manual and 4 for the assisted).

Manual Propagation

Subjects conducted two tasks: (1) identifying changes between the two versions from the documentation available in the *MediaWiki* web pages, and (2), adapting manually the transformation. The following equation resumes the main costs:

$$Manual\ Cost = D + P * \#Impacts$$

being D: the time estimated for detecting whether the new *MediaWiki* release impacts the transformation, P: the time needed to **P**ropagate a single change to the *MOFScript* code, and *#Impacts:* the number of instructions in the transformation **I**mpacted by the upgrade.

[7] http://ant.apache.org/

The experiment. D very much depends on the documentation available. For *MediaWiki*, designers should check the website[8], navigate through the hyperlinks, and collect those changes that might impact the code. The experiment outputted an average of 38' for $D_{MediaWiki}$, which is not very high due to the subjects being already familiarized with the *MediaWiki* schema. Next, the designer peers at the code, updates it, and checks the generated code. Subjects were asked to provide a default value for the newly introduced columns. On average, this accounts for 4' for a single update (i.e. P_{BRC} = 4'). Since the 1.17 upgrade impacted 3 *MOFScript* instructions, this leads to a total cost of 50' (i.e. 38 + 4*3). The execution time is considered negligible in both the manual and the assisted options since it is in the order of seconds.

Assisted Propagation

Subjects conducted two tasks: (1) configuration of the batch that launches the assisted adaptation, and (2), verification of the generated SQL script. The batch refers to a macro that installs the new DB release, injects the old schema and new schema, obtains the difference model, and finally, executes the adapter-aware *MOFScript* code. This macro is coded in *Ant* and some shell script commands. Running this macro outputs a *MOFScript* snippet along the lines of the new DB schema. Designers should look at this upgraded code since some manual intervention might still be needed. For instance, the introduction of new columns might also involve the assignment of a value that might not coincide by the one assigned by the adapter. Likewise, column deletion, though not impacting the transformation as such, might spot some need for the data in the removed column to be moved somewhere else. Worth noticing, the designer no longer consults the documentation but relies on the macro to spot the changes in the *MOFScript*. It is assumed that the comments generated by the adapter are expressive enough for the designer to understand the

[8]http://www.mediawiki.org/wiki/Manual:Database_layout

change (Listing 5.4). On this basis, the designer has to verify the proposed adaptation is correct, and amend it, if appropriate. The following equation resumes the main costs:

$$Assisted\ Cost = C + V * \#Impacts$$

being C: the time needed to **C**onfigure the batch; V: the time needed to **V**erify that a single automatically adapted instruction is correct and to alter it, if applicable.

The experiment. It took an average of 5' for the subjects to configure the macro (mainly, file paths) to the new DBMS release. As for V, it took an average of 6' for users to check the *MOFScript* code. Therefore, the assisted cost goes up to 11'.

Similar studies where conducted for other *MediaWiki* versions. Figure 5.5 depicts the accumulative costs of keeping *WikiWhirl* and *MediaWiki* in sync. Actually, until version 1.16 all upgrades were handled manually. Ever since, it is resorted to the macro to spot the changes directly on the generated *MOFScript* code. Does the effort payoff? Figure 5.5 shows the breakeven. It should be noted that subjects were already familiarized with the supporting technologies. This might not be the case in other settings. Skill wise, the manual approach is more demanding on *MOFScript* expertise while it does not require *Ant* knowledge. Alternatively, the assisted approach requires some knowledge about *Ant* but users limit themselves to peer at rather than to program *MOFScript* transformations.

The cost reduction rests on the existence of an infrastructure, namely, the adapter and the macro. The adapter is domain-agnostic, and hence, can be reused in other domains. On these grounds, we do not consider the adapter as part of the development effort of the *WikiWhirl* project in the same way that DBMS drivers are not included as part of the cost of application development. However, there is a cost of familiarizing with the tool, that includes the configuration of the batch macro (e.g. DB settings, file paths and the like). We estimated this accounts for 120' (reflected as the upfront investment for the assisted approach in Figure 5.5). On these

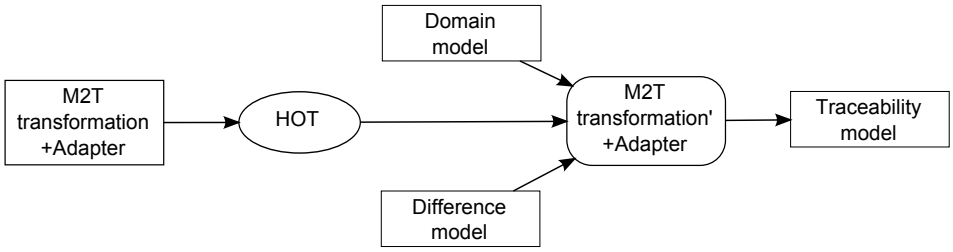

Figure 5.6: Generation of traceability model from the adapter

grounds, the breakeven is reached after the third release. To compute the profitability of the approach for another platform, it is suggested to apply specific constant values (D, P, V) to cost equations.

5.9 Dump Changes to the Transformation

When several evolution updates are accumulated in the platform, or when there is a big evolution including mayor changes, it will probably be a good strategy to dump to the transformation the changes done to the code in the adaptation. At some point, the developer will decide to transfer the changes done by the adapter to the transformation itself. This dump of the changes to the transformation is done manually, so as a help it is generated a record that contains information about what change have to be done to the transformation and where (in what line and column).

This record is a file, created by the adapter, that outputs the following information for each change: change in the platform (e.g. new column "column_name"), position (line and column) of the affected statement in the transformation and the change that has to be done. For instance:

```
1 #Added columns cl_type, cl_sortkey_prefix and cl_collation
2 #transformation line: 12, column: 11
3 INSERT INTO categorylinks (cl_from, cl_to, cl_sortkey,
     cl_timestamp, cl_type,
4 cl_sortkey_prefix, cl_collation) VALUES (@pageId,
     Softwareproject,
5 House_Testing, (DATE_FORMAT(CURRENT_TIMESTAMP(),
```

```
6 %Y%m%d%k%i%s), page, , 0);
```

In order to generate this record, the adapter have to know the position of each statement in the transformation. This is achieved adding the line and column to each of the prints using a HOT: this information is available in the model of the transformation. For example, *print("select * from ...")* is coverted into *printSQL("select * from ...", line, column)*.

The output of this impact analysis, apart from the aforementioned textual log, will be a traceability model that relates affected code with transformation elements generating it. Details on this traceability model will be given in the next chapter, where it will be the input of a tool that visualizes it.

5.10 Generalization of the Approach

Although the approach has been applied in a specific transformation language and with a specific platform, it can be applied to other scenarios as far as some premises are fulfilled:

- In order to compare different platform versions, platforms have to be injected into models. And in order to inject platforms into models, it is needed that the platforms are somehow specified. Some kind of formalization is needed in this specification to make possible the injection. For instance, in REST APIs there is not a formalized description, only documentation in natural language, making it difficult to automatize its injection.

- Statements to be adapted have to be independent from each other. In our example, the adaptation done in one statement does not affect the others. In other platforms, as APIs, a change in one statement might affect consequent statements.

- In order to dump changes to the transformation, as commented in 5.9, the model-to-text transformation itself has to be transformed. To

do this, a metamodel of the transformation language and an injector are needed.

5.11 Related Work

Target artifact/ Propagation techniques	Application	Data Schema / Data	Transformation
API	[CBH10]		
View	[Ra04]		
Rewriting	[MCD+08, CMDZ13]	[CMDZ13]	
Wrapper	[HH06, CH05]		This chapter

Table 5.4: The Design Space: synchronization approaches between the database schema and the affected artifacts.

This section phrases this chapter within the abundant literature in database schema evolution. The aim is not providing an exhaustive list but just some representative examples that serves to display the design space. Works on schema co-evolution can be classified along two dimensions: the target artifact to be co-evolved, and the approach selected to do so (see Table 5.4). As for the former dimensions, the target artifact includes:

- Instance data: Upgrading the schema certainly forces to ripple those changes to the corresponding data (e.g. tuples). This is referred to as "database co-evolution". An approach to automate this process is described in [CMDZ13]. In a similar vein, Dominguez et al. [DLRZ08] study the propagation of changes in the conceptual schema (specifically, an ER schema) to the database schema and the simultaneous co-evolution of the database. In this case, the co-evolution is realized through database triggers (a.k.a. event-condition-action rules) [PD99]). Finally, Cicchetti et al. [CRIP13]

look at how model changes (specifically, *WebML* models) can be propagated through model-to-text transformations to the supporting database. The scenario presented in this chapter is just the other way around: how changes in the database schema can be propagated to the model-to-text transformation.

- Applications: Artifacts consuming data are dependent upon the structures keeping this data. These artifacts include relational views [CMDZ13], data mappings (i.e. describing how data instances of one schema correspond to data instances of another) [VMP04] or application code [CH05]. This chapter focuses on a specific kind of applications: model-to-text transformations. This begs the question of what makes these transformations different from traditional applications. This moves us to the next paragraph.

- Transformation: If the artifacts consuming data are not directly coded but generated, then, co-evolution can be handled at the generation stage. Rather than directly changing the application code, schema changes can be propagated to the model-to-text transformations that end up producing the application code. Despite the increasing importance of model-driven generative techniques, this dimension has been mostly overlooked. This might be due to the understanding that traditional co-evolution techniques can be re-used for transformations as well. However, existing strategies to application co-evolution assumes that either the trace for database changes are known (and hence, can later be replicated for the application) or the SQL script is static (e.g. not obtained through a *prepare* statement)[CH05]. However, none of these premises hold in this chapter's scenario. First, the transformation and the platform are developed by two independent organizations. Second, the transformation do not *hold* but *construct* SQL scripts. The generative process is guided by the transformation code along the input model. Indeed, the SQL script is dynamically generated

once references to the input model are resolved. This specificity of model-to-text transformations makes it necessary to do the adaptation dynamically after the transformation engine has resolved the references but before it prints the result to the output file. As a result, solutions proposed for "static" scenarios are hardly applicable in this context.

The second dimension tackles the mechanisms used to approach the aforementioned co-evolution scenarios, namely:

- API: This technique allows programs to interface with the database through an external conceptual view (i.e. the API) instead of a logical view (i.e. the SQL data-manipulation language over the database schema). This technique is investigated in [CBH10] where they introduce an API which aims at providing application programs with a conceptual view of the relational database. The mapping between the conceptual schema and the class hierarchy of the API is as follows. Each entity type E belonging to the conceptual schema corresponds to a public Java class E in the generated API. Each attribute A of E corresponds to an instance variable A of class E. The API allows to navigate the data. For instance, the expression *"E2.getE1ViaR()"* recovers the instance of E1 associated with a given instance of E2 through R.

- Rewriting methods: They replace sub-terms of a formula with other terms. In our setting, this formula can stand for a database view or data mapping expression (a.k.a. Disjunctive Embedded Dependencies). A rewriting algorithm (e.g. the chase-based algorithm) reformulates a query/mapping upon an old schema into an equivalent query/mapping on the new schema. An example is described in [CMDZ13].

- Database views: Views are used to ensure logical data independence whereby the addition or removal of new entities, attributes, or

relationships to the conceptual schema should be possible without having to rewrite existing application programs. Unfortunately, solutions based on views cannot support the full range of schema changes. In particular, view mechanisms cannot simulate capacity-augmenting schema changes ("A schema transformation T is a total mapping, T : S -> S with S a set of schemas. T is capacity-augmenting if there exists a total and injective mapping f: states(S) -> states(T(S))" defined in [RLR98]). This matter is addressed in [Ra04]. For a *non*-capacity-augmenting schema change operation, this work derives a view schema intended by the operation. For a capacity-augmenting schema change operation, this work proposes that (1) the schema to be directly modified for the additional capacity required by the operation, (2) the original schema to be reconstructed as a view based on the modified schema, and finally (3), the target schema to be generated as a view also based on the modified schema [Ra04].

- Wrapper/Adapter: Wrappers are software that contains other data or software, so that the contained elements can keep existing in the newer system. They somehow support the inverse of views. Rather than "encapsulating" the new schema with the appearance of the old schema through a view, wrappers encapsulate the existing applications from the newer system. Wrappers for reusing legacy software components are discussed in [Sne00] and [THHB06], where they abound on the benefits of this approach. Back to databases, Cleve and Hainaut [CH05] introduce a wrapper approach where the new schema is encapsulated through an API. The wrapper converts all legacy database requests issued by the legacy programs into requests compliant with the new database.

5.12 Conclusions

The original contribution is to address, for a specific case study, the issue of transformation co-evolution upon DB schema upgrades. The suitability of the approach boils down to two main factors: the DB schema stability and the transformation coupling (i.e. the number of SQL instructions in the *MOFScript* code). If the DB schema stability is low (i.e. large number of releases) and the transformation coupling is high, the cost of keeping the transformation in sync, increases sharply. In this scenario, it is advocated for a preventive approach where the transformation is engineered for schema stability: *MOFScript's 'print'* is substituted by the adapter's *'printSQL'*. The adapter, using general recovery strategies, turns SQL statements based on the *old* schema into SQL statements based on the *new* schema.

Chapter 6

Testing MOFScript Transformations with HandyMOF

6.1 Introduction

Transformations need to be designed, programmed and tested. This last step becomes even more important if we consider that each transformation can potentially generate multiple applications, to which its errors would be propagated [SCD12].

Nevertheless, testing model transformation has proved to be a tough challenge [BGF+10]. Compared to program testing, model transformation testing encounters additional challenges which include the complex nature of model transformation inputs and outputs, or the heterogeneity of model transformation languages [TRL12]. To face this situation, both black-box techniques [BFS+06, FBMT09, SBM08] and white-box techniques [FSB04, GC12, KAER06] have been proposed. These two approaches are complementary and should be carried out in concert. As for the black-box technique, the challenge rests on coming up with an adequate set of input models. On the other hand, white-box techniques capture the mechanics

of the transformation by covering every individual step that makes it up [BGF+10]. This chapter concentrates on the latter, particularly focusing on *Model-to-Text (M2T)* transformations, which have received little attention.

The drawback of white-box testing approaches is that they are tightly coupled to the transformation language and would need to be adapted or completely redefined for another transformation language [BGF+10]. While there are standards [OMG11c] or well established languages [JABK08] in *Model-to-Model* transformation languages, the situation is more blurred in M2T transformations. This is the reason why, while aiming at the same goals as white-box testing (i.e., covering every step of the transformation), it has been opted to realize it using a mixed approach. The model test suite is generated using black-box techniques and then, both input models and the generated code are traced to the transformation. The purpose is twofold: (1) if a bug is detected in the generated code, it can be traced back to the transformation line that generated it, and (2) the transformation coverage obtained by the model test suite can be calculated based on transformation lines being transited.

Consequently, the approach heavily rests on trace models. Broadly, trace models need to capture a ternary relationship between the source model elements, the transformation model elements, and the generated code. It has been chosen MOFScript as the M2T transformation language as it already supports traceability between source model elements and locations in generated text files [OO07]. That is, it is possible to trace back the generated code from the source elements. Unfortunately, the third aspect (i.e. the transformation model elements) is captured at a coarse-grained granularity: the transformation rule. This permits coverage analysis to be conducted at the rule level (i.e. have all transformation rules been enacted?) but it fails to provide a deeper look inside rules' code. It would be similar to programming language testing stopping at the function calls without peering within the function body. Transformation rules might in themselves be complex functions where conditional statements and loops abound. Rule-based coverage might then fail to consider the

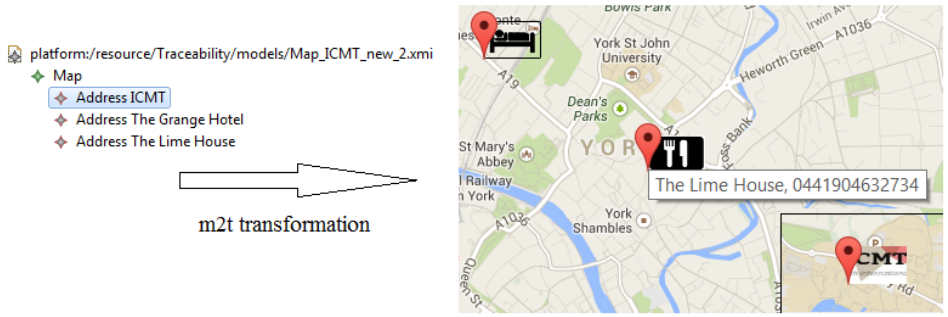

Figure 6.1: Input map model and desired output

diversity of paths which are hidden in the rule's body.

On these grounds, we complement MOFScript's native trace model with a second model that enables traceability between fine-grained transformation outputs (i.e. 'print' and 'println' statements) and locations in generated text files. An algorithm is introduced to aggregate trace models to ascertain which 'print' statements have not yet been visited during testing so that designers can improve their testing model suites to obtain full coverage. A transformation that generates *Google Web Toolkit (GWT)* code is used as the running example. These ideas are realized in *HandyMOF*, a debugger for MOFScript transformations. A video of HandyMOF at work is available[1].

6.2 Setting the Requirements

A common methodology for code testing generally comprises a number of well known steps: the creation of input test cases (i.e. the test suite), running the software with the test cases, and finally, analyzing the goodness of the results. Next paragraphs describe some of the challenges brought by transformation testing.

```
1 var index:Integer = 1;
2     ec.Map::main() {
```

[1] http://onekin.org/downloads/public/screencasts/handyMOF

```
3       [...]
4       f.println("public void onModuleLoad() {");
5       f.println("MapWidget map = new MapWidget();");
6       f.println("map.setSize(\"1000\", \"500\");");
7       f.println("map.setZoomLevel(14);");
8       ec.objectsOfType(ec.Address)->forEach(ecc:ec.Address)
            {
9         ecc.address();
10    }
11    f.println("RootPanel.get(\"mapContainer\").add(map);");
12    f.println(" }");
13  }
14  ec.Address::address(){
15    f.println("LatLng point"+ index +"= LatLng.newInstance(
          " + self.latitude + "," + self.longitude + ");");
16    f.println("MarkerOptions markeroptions" + index + " =
          MarkerOptions.newInstance();");
17    f.print("markeroptions" + index + ".setTitle(\"" + self
          .name + ", " + self.description);
18    if(self.description = "restaurant"){
19      f.print(", " + self.telephone);
20    }
21    f.println("\");");
22    f.println("Marker marker" + index + " = new Marker(
          point" + index + ", markeroptions" + index + ");");
23    f.println("LatLng sw" + index + " = LatLng.create("+
          self.latitude +","+self.longitude+");");
24    f.println("LatLng ne" + index + " = LatLng.create("+
          self.latitude +","+self.longitude +");");
25    self.pictures->forEach(pic){
26      f.println("LatLngBounds bounds"+index+" =
            LatLngBounds.create(sw"+index+", ne"+index+");");
27      f.println("GroundOverlay go"+index+" = new
            GroundOverlay(\""+  pic +"\", bounds"+index+");")
            ;
28      f.println("map.addOverlay(go" + index + ");");
29    }
30    f.println("map.addOverlay(marker" + index + ");");
```

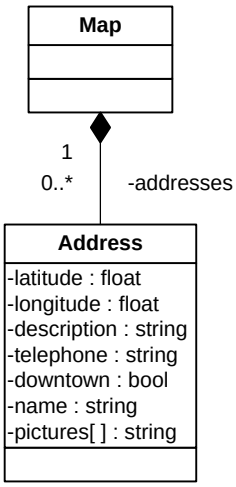

Figure 6.2: Map metamodel

```
31  index += 1; }}
```

Listing 6.1: Map2GWT transformation

Creation of test suites. Obtaining the appropriate *test suites* becomes critical to ensure that all the transformation variations are covered, and hence, representative code samples are obtained. So far, *Pramana* serves engineers to generate 'model suites' for 'metamodel coverage'. i.e. checking the full expressiveness of the metamodel [SBM08]. This certainly covers variations on the model structure, and it might be used as a black-box testing approach for transformations. However, Pramana does not guarantee that the generated samples cover all the branches of the transformation. This calls for Pramana to be complemented with a white-box testing approach where the unveiling of the transformation code provides additional input to obtain the test suite.

As an example, consider a transformation to generate markers in Google maps (see Figure 6.1). Markers represent *Points of Interest* (POI). A conference page contains the locations of the venue and the main hotels or restaurants available in the area. Those markers are captured through a *Map* metamodel (Figure 6.2). Transformation rules are defined to handle

the two elements of the *Map* metamodel, namely, *Map* and *Address*. The output is a Google map where markers are depicted together with their pictures, if available. Besides, if the marker stands for a restaurant, the phone is shown as part of the marker's content. This last rule illustrates the need for white-box testing. The significance of 'restaurant' as a key value for changing the transformation flow cannot be ascertained from the string-typed property 'place' in Pramana. Therefore, the use of metamodel-based test suite generators like Pramana does not preclude the need to check that all paths of the transformation have been traversed.

Analyzing the goodness of the results. In the testing literature, an oracle is a program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object [Bez90]. Oracles can be as simple as a manual inspection or as complex as a separate piece of software. This chapter is focused on assisting manual inspection. This requires means for linking code back to generators (i.e., MOFScript rules), and vice versa. MOFScript's native trace model provides such links at the rule level. However, a rule-based granularity might not be enough. The *address* rule (see Figure 6.1 - lines 14-31) illustrates how transformation complexity is tied to the complexity of the metamodel element to be handled or the logic of the transformation itself. This results in 'print' statements being intertwined along control structures such as iterators and conditionals. A rule-based granularity encloses the whole output within a single trace, failing to indicate the rule's paths being transited. A print-based granularity will account for a finer inspection of the transformation code. This in turn, can redound to the benefit of coverage analysis and code understanding. This sets the requirement for fine-grained traces.

6.3 The HandyMOF Tool

The previous section identifies two main requirements: semi-automatic construction of test suites, and fine-grained linkage between transformations and generated code. These requirements guide the

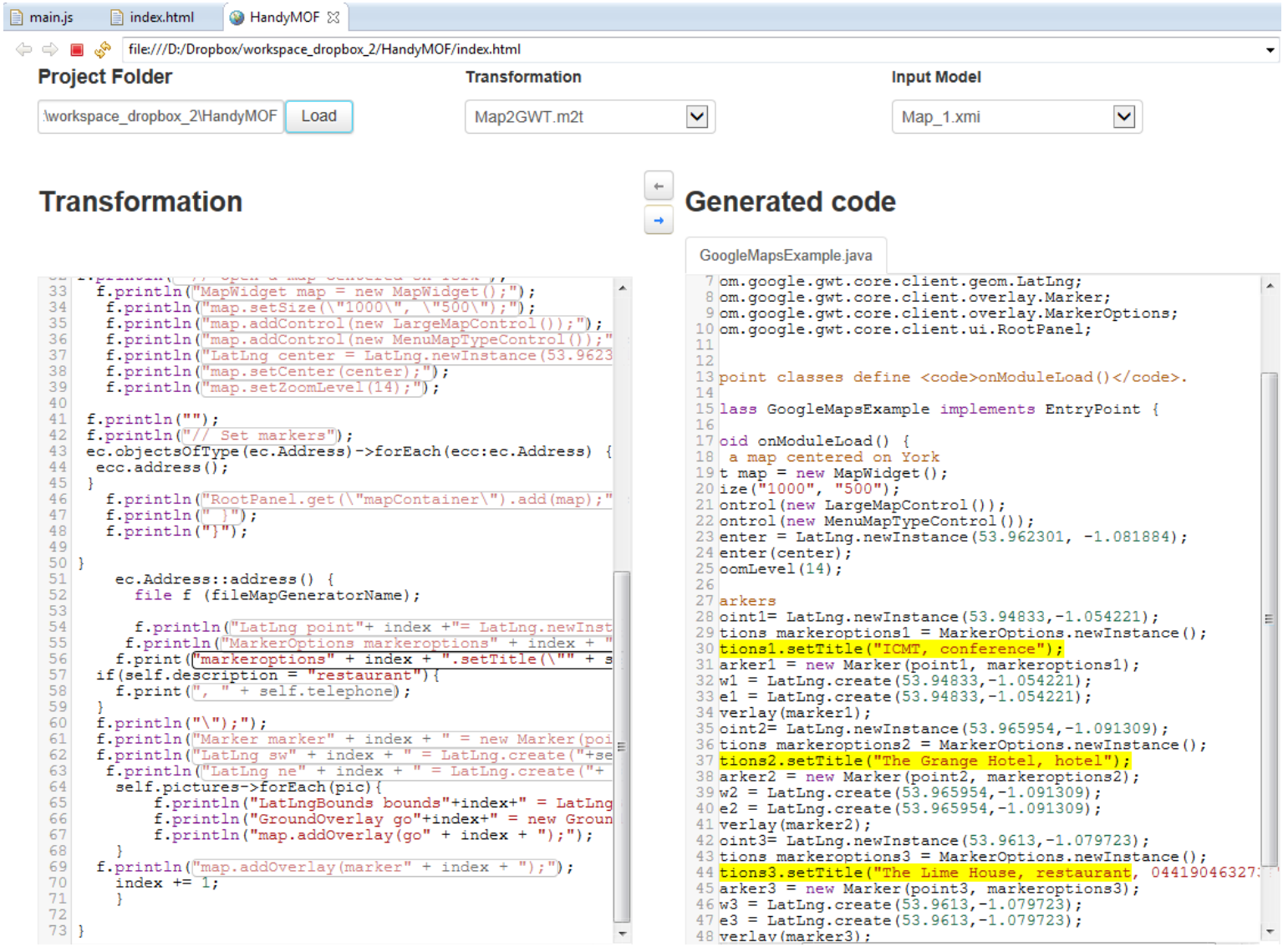

Figure 6.3: HandyMOF as a debugger assistant: from transformation to code

development of HandyMOF, a debugger for MOFScript included as part of Eclipse (see Figure 6.3). The canvas of HandyMOF is basically divided in two areas:

- *configuration area:* where the testing scenario is defined. This includes: (1) the project folder, (2) the transformation to be debugged (obtained from the *transformation* folder in the project), and (3), the input model to be tested (obtained from the trace models that link to the chosen transformation).

- *inspection area:* previous configuration accounts for a transformation enactment that can output one or more code files. The inspection area permits to peer at both the transformation and the code files. The output reflects a single transformation enactment (the one with the input model at hand). Figure 6.4 shows the case for the input model Map_1.xmi. In this case, only one code file is generated (i.e. GoogleMapsExample.java). Additional code files would have been rendered through additional tabs.

The added value of HandyMOF basically rests on two utilities. First, it permits to selectively peer at the generated code. To this end, both the transformation and the generated files are turned into hypertexts. Code is fragmented in terms of 'traceable segment' (i.e. set of characters outputted by the enactment of the same 'print', see later). Finally, both MOFScript print statements and 'traceable segments' are turned into hyperlinks. In this way, debugging answers are just a click away. Answers to questions such as 'which code does this print statement generate?' or ' which print statement caused this traceable segment?' are highlighted by just clicking on the respective hyperlink. Figures 6.3 and 6.4 illustrate two debugging scenarios. In Figure 6.3, it is inspected the output of a given 'print': it answers the question : which code snippet results from the enactment of this print?' Click on the print statement ('Transformation' textarea, line 56) and the answer is highlighted. In Figure 6.4 it is traced back a code snippet to its generator (i.e. 'print' statement), respectively. In this

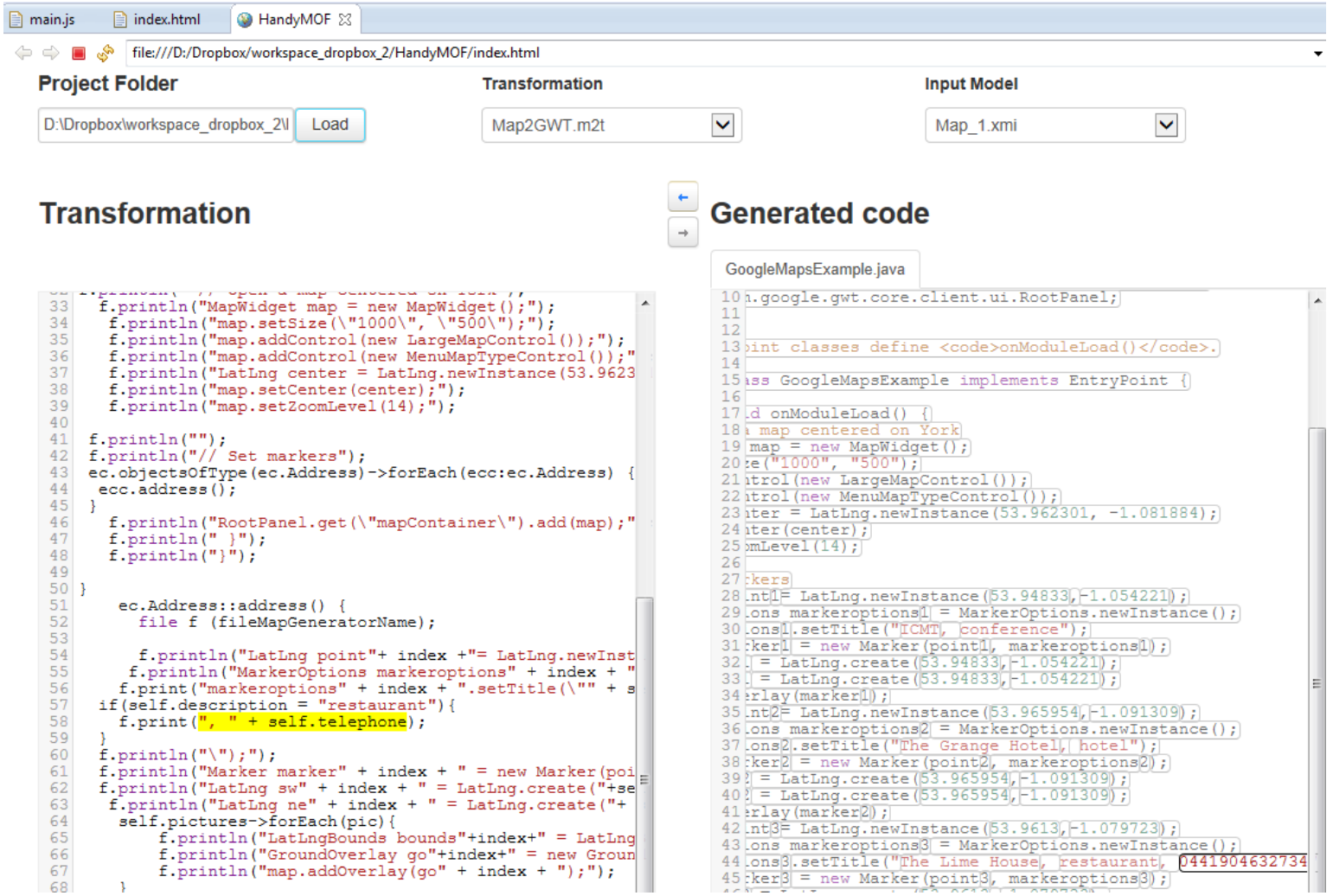

Figure 6.4: HandyMOF as a debugger assistant: from code to transformation

case, it is answered the question: which 'print' statement causes this code snippet? Click on the code snippet ('Generated code' textarea, line 44) and the answer is highlighted ('Transformation' textarea, line 58).

The second utility is the role of HandyMOF as a coverage analysis assistant. First, by identifying 'holes' in the Pramana generated model suite in terms of 'print' statements not yet visited by any input model. Second, by identifying the smaller set of model inputs that provides the larger coverage (see later), hence coming up with a *minimal model suite* which can speed up future testing. The process starts by selecting 'all' as for the input model configuration parameter (see Figure 6.5). This triggers the algorithm for the obtainment of the minimal model suite. The output is reflected in two ways. First, it renders the model identifiers of such suite in the right panel. Second, it aggregates the resulting trace models, collects the visited 'print' statements, and highlights in the left panel of the inspection area, those 'print' statements not yet transited. This helps developers to elaborate additional input models to increase transformation coverage.

6.4 The HandyMOF Architecture

Figure 6.6 depicts the main components and flows of HandyMOF. The Project Explorer handles the folder structure. Pramana provides input models from the corresponding metamodel. Finally, HandyMOF consumes input models and transformations to obtain its own trace models, that complement MOFScript's native ones, and the generated code files.

An important question is whether this approach can be generalized to other M2T transformation languages. Basically, HandyMOF rests on two main premises. First, the existence of a trace model that links the input model with the generated code. Second, the existence of a transformation metamodel (and the corresponding injector) that permits to move from the transformation text to its corresponding model, and vice versa. Provided these characteristics are supported, HandyMOF can be

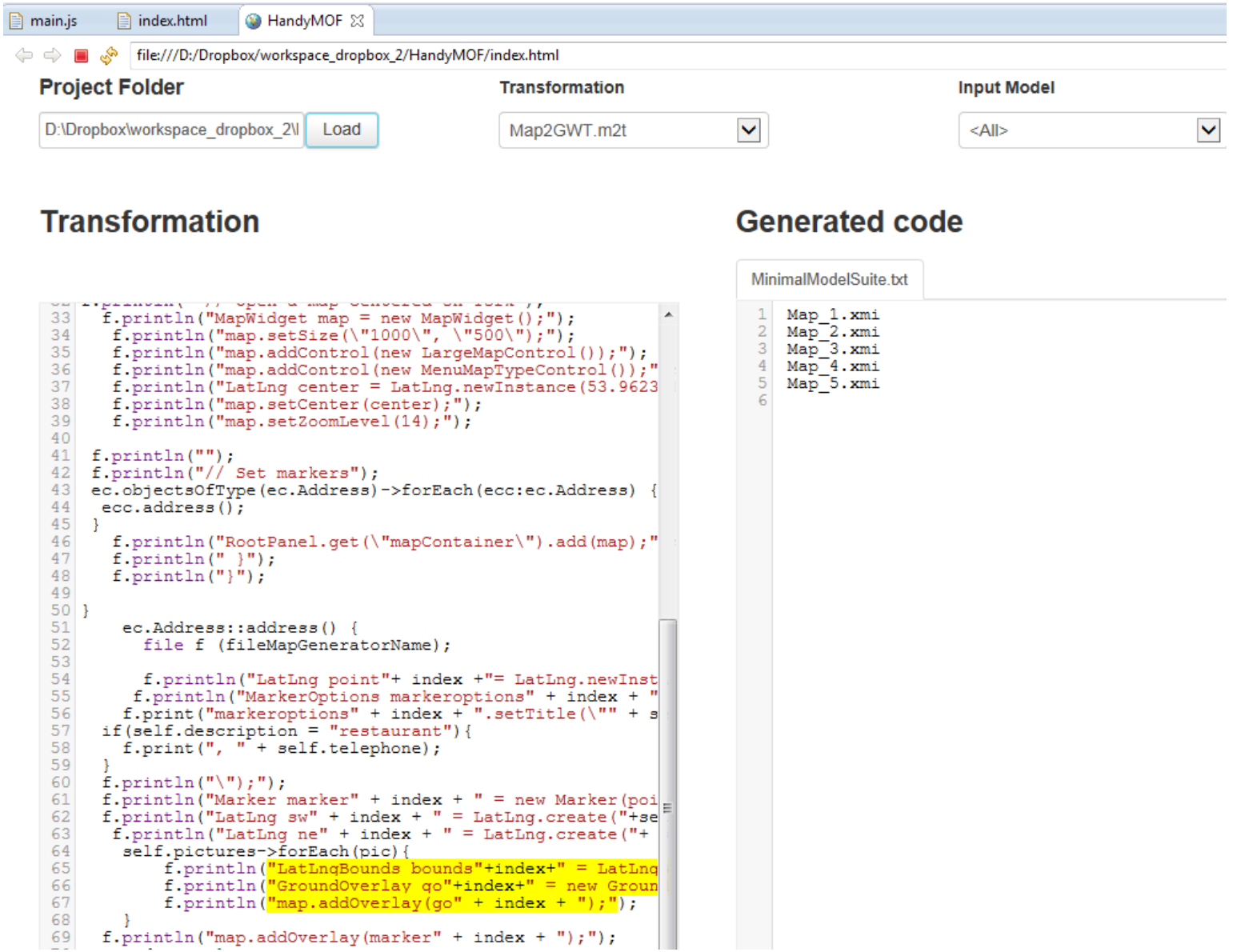

Figure 6.5: HandyMOF as a testing assistant.

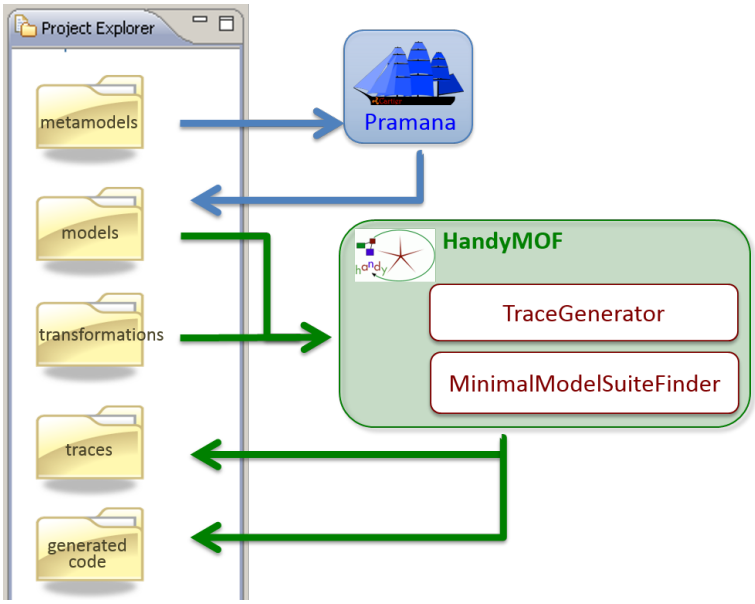

Figure 6.6: HandyMOF's Architecture

extended to languages other than MOFScript. Next subsections delve into the main components of HandyMOF, namely the *Trace Generator* and the *Minimal Model Suite Finder*.

6.4.1 Trace Generator

The goal of this component is to trace the input model, the generated code and the M2T transformation. It leverages on the trace natively provided by MOFScript that links the input model with the generated code. The metamodel for HandyMOF's traces is first described, followed by how these traces are generated.

HandyMOF's Trace Metamodel

MOFScript's trace metamodel defines a set of concepts that enable traceability between source model elements and locations in generated text files (see Figure 6.7 left) [OO07]. A trace contains a reference to the operation (transformation rule) that generated the trace and references

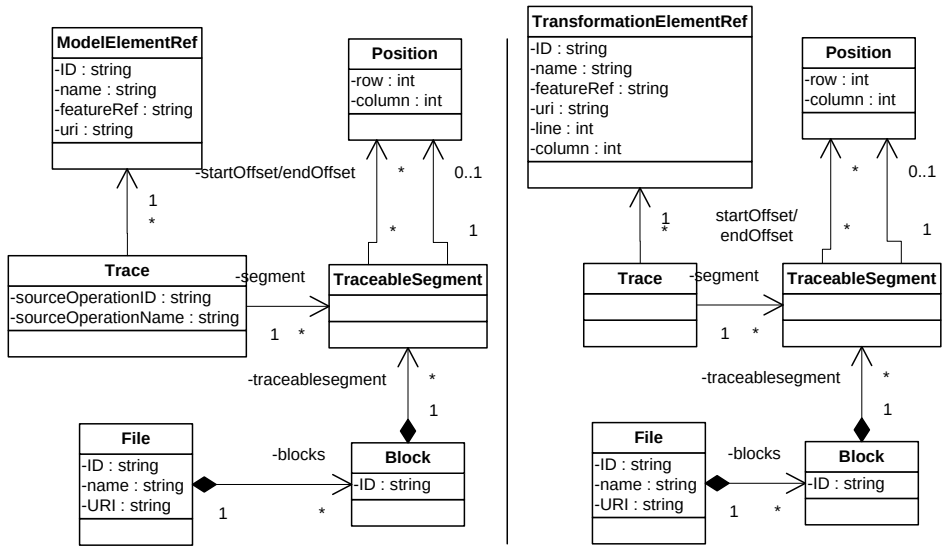

Figure 6.7: MOFScript's Traceability Metamodel (left, obtained from [SBM08]) and HandyMOF's trace metamodel (right)

the originating model element and the target traceable segment. The model element reference contains the 'id' and 'name' for the originating element. It also contains a feature reference, which points out a named feature within the model element (such as 'name' for a property class). On the other hand, the generated code file is captured in terms of 'blocks'. Blocks are identifiable units within a file. A block contains a set of segments which are relatively located within the block in terms of a starting and ending offset.

This metamodel nicely captures traces from source model elements to the generated code file through traceable segments. Unfortunately, traceable segments are related to their transformation rule counterparts rather than to the inner 'print' statements. We claim that a finer granularity might help a more accurate debugging in the presence of large transformation rules. On these grounds, we complement the natively provided MOFScript trace model with our own trace model where 'traceable segments' are linked back not just to transformation

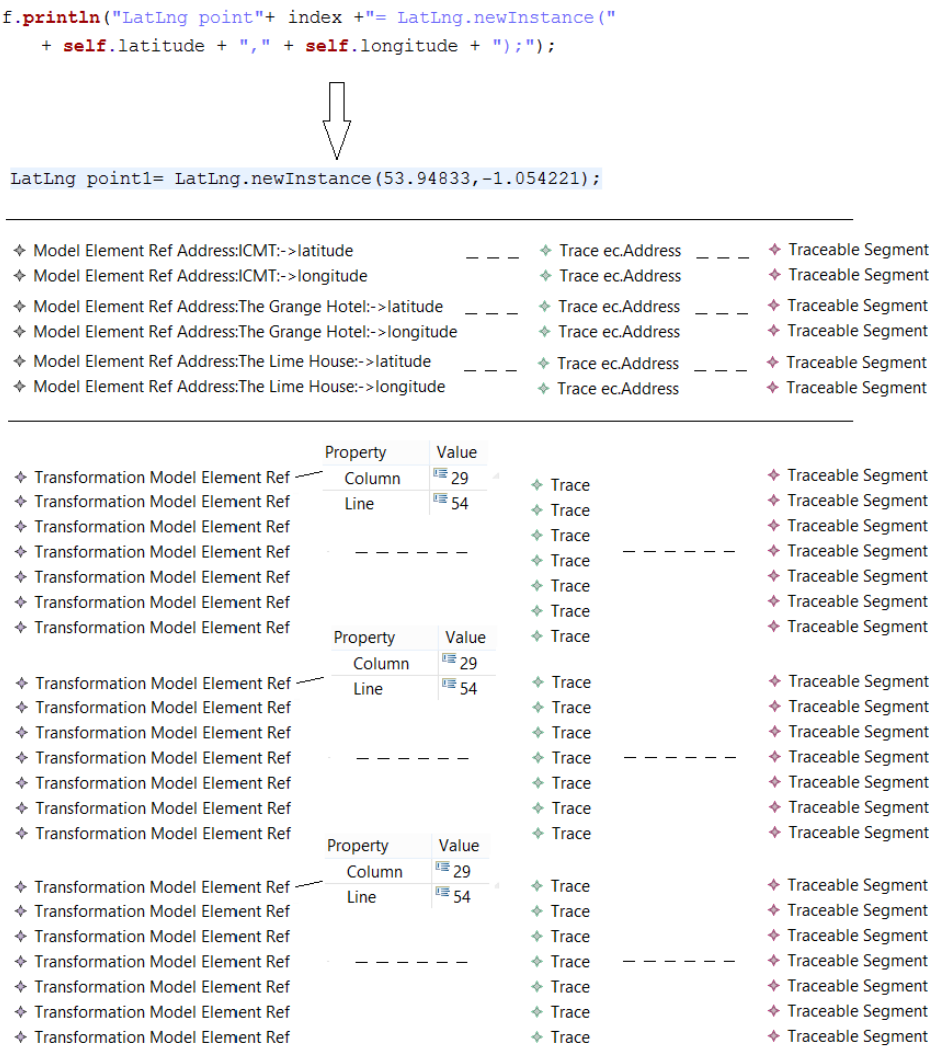

Figure 6.8: Complementary trace model: between model and code (above) and between transformation and code (below)

rules but to the transformation's 'print' statements. Figure 6.7 right depicts HandyMOF's trace model. Differences stem from the granularity of traceable segments. MOFScript traceable segments account for rule enactments. In HandyMOF, these segments are now partitioned into fine-grained segments: one for each enacted 'print' statement. Figure 6.8 illustrates the two complementary traces for a simple case: between model and code (above) and between transformation and code (below). In this case, as the 'println' is composed of seven parts, seven traces will be given, one for each. As the 'print' is executed three times (one to create a location for a conference, one for an hotel and the other for a restaurant), we can see that those traces are tripled. The position of the 'print' in the transformation to be the same, as captured in *TransformationModelElement*.

Obtaining Trace Models in HandyMOF

To trace a whole M2T transformation and going over all its branches, more than one target code is necessary. The first activity, ***Model Instance Generation***, creates the input models that permit to work out the code needed to get the corresponding traces. The goal would be to generate models that obtain a 100% coverage of the transformation. However, to the best of my knowledge, no tool exists that, given an input metamodel and a M2T transformation, generates the model instances that provide full coverage of the transformation. As a result, it is opted for using Pramana, a tool that implements black-box testing for metamodels [SBM08]. Using Pramana, the MDE Engineer generates a set of model instances that cover the input metamodel.

Once the set of models has been obtained, the next goal is to link the M2T transformation with the code generated from these metamodels. The generated code is plain text, so the trace model links the transformation

elements with the position where the related code fragment starts. This position can be different depending on the input model: it depends on execution flow. As a case in point, imagine an *if-then-else* statement in the transformation. Each branch may have a different number of 'print' statements. As a consequence, the position where the first statement after the 'if' starts may vary depending on the executed branch. The same holds for loops, depending on the input model they may be executed a different number of times thus changing the position where the rest of the statements start. Therefore, the trace model must be generated at runtime.

It must be created the trace between the M2T transformation and the code generated for every input model of the set. The approach to generate the trace model has been explained in the Section 5.9 of previous chapter:

- First, using a HOT, it is inserted in every print instruction its line and column in the transformation: *e.g. println("code to be printed")* -> *printtrace("code to be printed", 12, 15)*. This information is taken from the model representation of the transformation.

- Then, when executing the transformation, the print will not be directly executed, but will be redirected to a function (*printtrace*) that will generate the traces using the transformation positions of the parameters; and the physical positions in the code file, based on the length of the code to be printed. Figure 6.9 represents a trace model: it relates the position of transformation elements (*Trace* elements) with positions in the code (*TraceableSegment* elements).

To sum up, as a result of this process using techniques of black-box testing, we have created a set of input models, generated their corresponding code using the M2T transformation, and traced this transformation to each instance of code via the trace models.

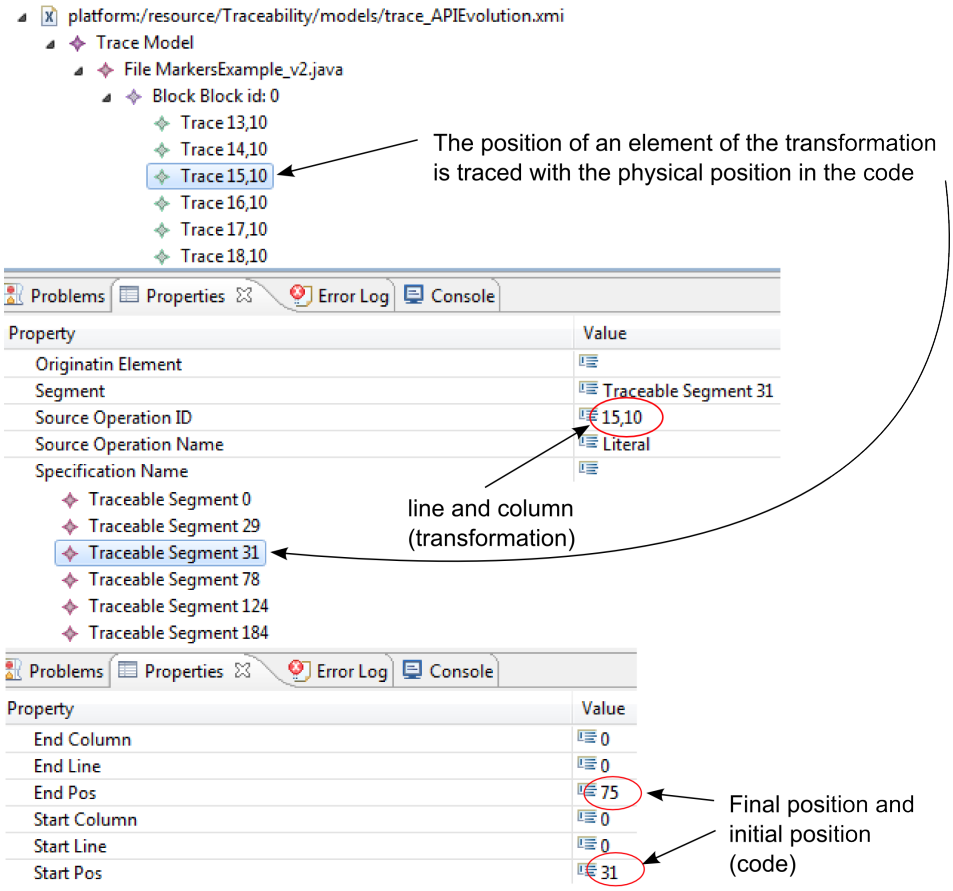

Figure 6.9: Trace between transformation and code

6.4.2 The Minimal Model Suite Finder

In order to analyze the M2T transformation and see to what extent its statements have participated in the code generation, the use of input models is unavoidable. As commented previously, we opted for using Pramana (formerly known as Cartier) [SBM08], a tool that implements black-box testing for metamodels.

Are models generated by Pramana enough to obtain our goal? Pramana serves engineers by generating *model suites* for *metamodel coverage* but its purpose is not *transformation coverage*. However, transformations have embedded semantics that need to be considered if the goal is the

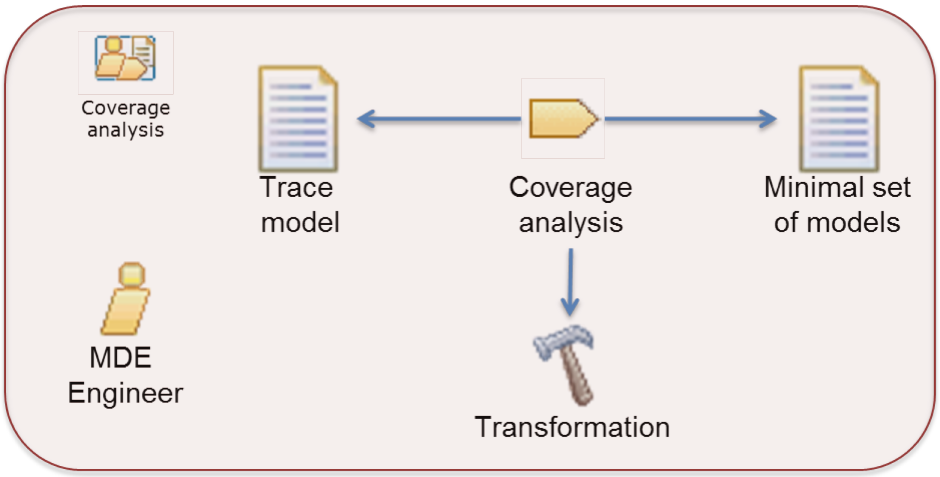

Figure 6.10: Coverage analysis

latter. Different conditions present in *if* statements or loops require specific test cases that may not be generated if the criteria is merely metamodel coverage. As a case in point, the *if* statement in Figure 6.1 (line 18) checks whether the *Address* corresponds to a restaurant. Among the many test cases that can be generated from the metamodel, this statement requires one with precisely that value in the *description* attribute to obtain *transformation coverage,* which is not guaranteed if the generation of the test cases does not take the transformation into account. Hence, as in program testing where black-box testing and white-box testing approaches are used in concert, we need to cater for both metamodel and transformation coverage.

The proposal of this work is the use of trace models for the analysis of *transformation coverage*. What is needed is to link the code samples with the transformation, via the tracing models obtained by the trace generator module. It is needed to be checked how much coverage has been reached using the input models generated by Pramana.

Hence, the task of the *MinimalModelSuiteFinder* module (see Figure 6.6) is to quantify the transformation coverage, and to rule out those input models whose transformation only enacts transformation statements that

have already been traversed by previous models. The goal of the module is then to obtain the minimal set of input models that get the higher coverage percentage of the transformation code (specifically, the 'print' instructions that generate the target code). This set is called the *minimal model suite*. Finding a global optimum is NP-hard, so the presented greedy algorithm ensures a first solution.

```
1  helper def: getModels (availableModels: Sequence(Trace!
       TraceModel),
2    minimalModelSuite: Sequence(Trace!TraceModel),
3    coveredPrints: Sequence(String)): Sequence(Trace!
       TraceModel)=
4    minimalModelSuite->append(availableModels->select(e|self.
       bestModel(e, availableModels, coveredPrints))->first
       ()),
5    coveredPrints->append(availableModels->select(e|self.
       bestModel(e, availableModels, coveredPrints))
6      ->first().trace->collect(e|e.line))->flatten())
7    if coveredPrints.size() = self.numberOfLines or
       availableModels.size()=0 then
8      minimalModelSuite
9    else
10     self.getModels(availableModels->excluding(
         availableModels->select(e|self.bestModel(e,
         availableModels,
11     coveredPrints))->first())
12   endif
13 ;
```

Listing 6.2: Minimal model suite algorithm (main rule)

Figure 6.2 shows one of the functions of the algorithm used in obtaining this suite. It is a recursive function that finishes when all lines are covered or there are not more input models to use (line 7). The algorithm can be summarized as follows:

1. The best model is added to the list of selected models (*minimalModelSuite*) (line 4).

2. The prints covered by the best model are added to the list of covered prints (*coveredPrints*) (lines 5-6).

3. The best model (i.e., the one that covers most prints) is excluded from the available models (*availableModels*) (lines 10-11).

Using these two modules the interface of HandyMOF can be used to check the correspondence between the M2T transformation and the generated code, be it on a single instance (see Figure 6.4) or for the complete model suite to check the obtained coverage (see Figure 6.5).

6.5 Related Work

This chapter is grounded on the fact that black-box and white-box testing techniques are complementary. Black-box testing approaches do not capture the mechanics of the transformation [BGF+10], which is precisely where it is intended to aid. McQuillan et al. propose white-box coverage criteria for transformations [MP09]. Although their work centers in ATL [JABK08] (i.e., a model-to-model transformation), their coverage criteria could be applicable to the presented case as well. This chapter focuses on instruction coverage (more precisely on coverage of instructions that produce an output in the generated code). Gonzalez et al. present a white-box testing approach for ATL transformations [GC12]. It follows a traditional white-box testing strategy where input models are created based on the inner structure of the transformation. This involves a coupling between the approach and the transformation language. While this makes sense in a model-to-model setting, where ATL has become de facto standard, there is no predominant language for model-to-text transformations. This is why it has been opted for a mixed approach where the input models are generated using black-box testing (i.e., it makes the model generation independent from the language). The approach, albeit less precise, can be applied with any language with which can be obtained or builded the adequate traces.

Currently no standard or well established proposals exist for model-to-text transformation testing [TRL12]. Wimmer et al. present an extension of *tracts* [GV11] to deal with model-to-text transformations [WB13]. Their approach is complementary to the one presented here, as it focuses on black-box testing (i.e., it considers the specification of the transformation, not its implementation).

Regarding transformation debugging, Sun et al. present a proposal for debugging demonstration-based model transformations [SG13]. This chapter also focus on debugging, but for the case of model-to-text transformations.

6.6 Conclusions

This chapter presented a proposal for white-box testing of M2T transformations. Due to the heterogeneity of M2T transformation languages, the test suite is generated using black-box testing and then, the generated code is traced back to the transformation and the input model. Main outcomes include: (1) if a bug is detected in the generated code, it can be traced back to the generating print statement, (2) each generator statement (i.e., 'print') can be traced to the generated code line, and (3) the transformation coverage obtained by the test model suite can be calculated in terms of visited 'prints'. If the obtained coverage is not full, the developer can create input models that cover the missing transformation lines. This is realized in HandyMOF, a tool for debugging MOFScript transformations.

Of note, this proposal could be generalized for any transformation language fulfilling our both premises, namely, the existence of a transformation metamodel (and its injector) and a trace model linking the input model with the generated code.

Chapter 7

Conclusions

7.1 Overview

MDE claims many potential benefits, one of them maintainability. But the truth is that there is no empirical evidence that this is indeed the case [VWVDVD07]. Software maintenance accounts for the most of the total production cost, and this worsen for the MDE case. MDE ecosystem is heterogeneous and complex, and artifact dependencies between actor need to be kept in sync. This work is focused on protecting the investment done in the MDE infrastructure, offering techniques and tools that ease the maintainability of MDE artifacts. Specifically, it is concerned with the evolution forces affecting metamodels and technological platforms .

In this chapter, the main contributions are summarized, their limitations are discussed and some open research questions are proposed as future work.

7.2 Results

The contributions of this Thesis are done in three of the central chapters. A recapitulation of these contributions are exposed next.

7.2.1 Contribution 1

Contribution. Chapter 4 proposes a semi-automatic approach to co-evolve model-to-model transformations to metamodel evolution. The process is composed of the following steps: (1) detecting simple changes between the original and the evolved metamodel, (2) deriving complex changes from simple changes, (3) translating boolean expressions to the CNF form, (4) use metamodel matching between input and output metamodels to look for similarity, and finally, (5) adapt the original transformation into an evolved transformation that conforms to the new version of the metamodel.

Proof-of-concept. The approach is realized for EMOF/ Ecore-based metamodels, and ATL transformations. Two are the main artifacts of the prototype: a transformation that transforms simplex changes into complex changes and a HOT that adapts the transformation to changes.

Dissemination & contrast with the community. Jokin García, Oscar Díaz, Maider Azanza. "Model Transformation Co-evolution: a Semi-automatic Approach". In Software Language Engineering (SLE), 2012, Dresden, Germany.

7.2.2 Contribution 2

Contribution. Chapter 5 tackles the impact of platform evolution on model-to-text transformations. The strategy used to avoid the desynchronization of transformations when the target platform evolves, is to use the adapter pattern to adapt the generated code upon platform upgrades. This approach has been tested in a particular scenario: using databases as platform. The adapter, using general recovery strategies, turns SQL statements based on the *old* schema into SQL statements based on the *new* schema.

Proof-of-concept. The adapter is supported as a library of the transformation. A batch script automatizes the whole process, including the injection and comparison of the platforms and execution of the transformation. An empirical evaluation has been done to assess the

profitability of the approach.

Dissemination & contrast with the community. Jokin García, Oscar Díaz, Jordi Cabot. "An Adapter-Based Approach to Co-Evolve Generated SQL in Model-to-Text Transformations". In International Conference on Advanced Information Systems Engineering (CAiSE), 2014, Thessaloniki, Greece.

7.2.3 Contribution 3

Contribution. Chapter 6 presents a tool that helps the developers to debug model-to-text transformations and to assess the completeness of the input model suite. The problem with black-box testing, is that it does not guarantee the transformation coverage. To help with this problem, the tool does a transformation coverage analysis. If the obtained coverage is not complete, the developer can create input models that cover the missing transformation lines. Moreover, the tool allows the debugging of the transformation: if a bug is detected in the generated code, it can be traced back to the generating print statement; and each generator statement (i.e., 'print') can be traced to the generated code line.

Proof-of-concept. This is realized in HandyMOF, a tool for debugging MOFScript transformations. This tool visualizes graphically the relationship between transformation and its generated code.

Dissemination & contrast with the community. Jokin García, Maider Azanza, Arantza Irastorza, Oscar Díaz. "Testing MOFScript transformations with HandyMOF". In International Conference on Model Transformations (ICMT), 2014, York, United Kingdom.

7.3 Research visits

In a globalized world, research communities are not limited by physical distances. Researchers around the world put their work in common, criticize each other and collaborate no matter which their origin is. As

a commitment with this internationalist vision of research, I wrote my contributions in English and presented them in international conferences. Another activity considered as beneficial for a PhD candidate, is to conduct a research stage in a foreign country. In this case, I visited the *AtlanMod* group in the *Ecole des Mines* of Nantes, in France, under the supervision of Dr. Jordi Cabot, from June to August of 2012. The benefits of the visit has been at least two. On one hand, being with unfamiliar experts, makes you learn about different topics, or realize about different perspectives of the topics you are familiarized with. On the other hand, it makes you learn about different working habits, methodologies and ways to organize and interact.

7.4 Future research

A Thesis tries to resolve a problem. Nevertheless, in its development, some issues remain in the pipeline. Then, I expose some of the open research questions.

When facing the maintainability of a MDE infrastructure, it is primordial to know the differences between various versions of a model. When comparing code versions, textual comparison can be valid. But in the case of models, the comparison must be structural, being the difference in terms of changes between elements and relations (addition, deletion or update). The most used metamodel-agnostic comparison tool, is EMFCompare. In spite of the complexity of comparing models (NP-complete), I have found that the tool is effective and efficient in many scenarios. Specifically, in the scenarios used in this Thesis (metamodel comparison and database model comparison) it works as expected. The problem is that for other scenarios where the metamodel to be compared is very complex, the effectiveness decreases. The use of UUIDs (unique identifiers) in models improves dramatically the comparison in this cases, but I foresee a need for enhancing generic comparison to the specific domain of the model.

In chapter 4, the prototype has been tested for an example as a proof of concept. But, obviously, further experimentation with real scenarios would be advisable. The difficulty here is to find a real metamodel that evolves and that is used in a real transformation.

In chapter 5, one of the obvious pending task is to generalize the approach to other platforms, such as APIs. APIs are also the subject of frequent changes which tend to be out of the control of the programmers. When this code is generated from models, adapter-aware transformations can also be valuable in this setting. And going beyond the specific scenarios, I wonder whether it would be possible to abstract from the specific platform and study it from a generic viewpoint, injecting a model conforming *Abstract Syntax Tree Metamodel* (*ASTM*) [1], which could abstract from different languages or even *Knowledge Discovery Metamodel (KDM)*[2] to deal with platforms which are not only based on code. Therefore, instead of using one specific metamodel for each platform type, a standard metamodel could be used to describe generically all platform scenarios. And when generalizing this problem, it is needed to define a generic methodology to develop adapters for the producer of the adapters.

Finally, in chapter 6, it arises the need of guiding transformation developers in creating the missing input models from the unvisited 'prints'. The algorithm to detect the minimum set of models in the coverage analysis is also improvable. It is also contemplated integrating HandyMOF with other testing approaches to provide an integrated solution. And last, an exhaustive experiment with real transformations is pending. The difficulty is to find transformation developers so they can assess the usefulness of the tool, and to find big real model-to-text transformations, where the use of this kind of tools makes sense.

Apart from these short-term improvements focused on overcoming limitations of presented approaches, I also would like to envisage some long-term considerations about transformation maintainability:

[1] http://www.omg.org/spec/ASTM/1.0/
[2] http://www.omg.org/spec/KDM/1.0/

- The same way they exist patterns and good practices in imperative programming, after some years of many people using transformations, it is time to collect that knowledge from the experts. The best way to improve the maintainability of model-to-model code would be to follow guidelines from experts.
- There exist a gap between "model world" and "text world" that is produced when executing the model-to-text transformation. There are tools that can inject the generated code into models, but in my opinion, the transformation must be "aware" of what kind of code is generating, and not consider it as plain text.

Appendix A

M2M Transformation Adaptation

In Chapter 4, the adaptation of the transformation is done using a HOT. This appendix shows one of that adaptations as example.

```
1  helper def : DiffSize : Integer = 6;
2  helper def : sourceMM : String = 'ExamXML';
3  helper def : targetMM : String = 'AssistantMVC';
4  helper def : index : Integer = 1;
5  helper def : indexBinding : Integer = 1;
6  helper def : getUpdateAttributeLeft : Sequence(String) =
7    DIFF!ComplexChange.allInstances()->select(e|not e.
       oclIsUndefined())->asSequence()
8    ->select(e|e.changeType=#Split_class)   ->collect(e|e.
       atomicChanges)->first()->select(e|e.oclIsTypeOf(DIFF!
       UpdateAttribute))  ->collect(e|e.leftElement.toString
       ().split('!')->last());
9  helper def : getRemoveModelElementLeft : Sequence(String) =
10   DIFF!ComplexChange.allInstances()->select(e|not e.
       oclIsUndefined())->asSequence()
11   ->select(e|e.changeType=#Split_class)   ->collect(e|e.
       atomicChanges)->first()->select(e|e.oclIsTypeOf(DIFF!
       RemoveModelElement))   ->collect(e|e.leftElement.
       toString().split('!')->last());
12 helper def : getSplittedClasses : Sequence(String) =
```

```
13    self.getUpdateAttributeLeft;
14  --splitted class in the source
15  helper def : deletedBindings : Sequence(ATL!Binding) =
16    ATL!Binding.allInstances()->select(e|e.value.oclIsTypeOf(
        ATL!NavigationOrAttributeCallExp))    ->select(e|e.
        value.name=self.getRemoveModelElementLeft->first());
17  helper def : getAttributesFromClass (param : String) :
      Sequence(String) =
18    Ecore!EClassifier.allInstances()->select(e|e.name=param)
         ->collect(e|e.eAllAttributes)->first()->collect(e|e.
        toString().split('!')->last());
19  helper def : getAttributesFromClass2 (param : String) :
      Sequence(String) =
20    Ecore2!EClassifier.allInstances()->select(e|e.name=param)
           ->collect(e|e.eAllAttributes)->first()->collect(e|
        e.toString().split('!')->last());
21  --difference between attributes from original and splitted
22  --class in param
23  helper def : deletedAttributes (param : String) : Sequence(
      String) =
24    self.getAttributesFromClass(self.getUpdateAttributeLeft->
        first())    ->asSet().symetricDifference(self.
        getAttributesFromClass2(param)->asSet());
25  --returns if param1 is in param2
26  helper def : containsString (param1 : String, param2 :
      Sequence(String)) : Boolean =
27    param2->iterate(p; y : Boolean = false |
28      if p = param1 then
29        true
30      else
31        if y = true then
32          true
33        else
34          false
35        endif
36      endif    )    ;
37  helper def : deletedRule : Sequence(ATL!MatchedRule) =
38    ATL!MatchedRule.allInstances()->
```

```
39    select(e|self.containsString(e.inPattern.elements->first
          ().type.name, self.getSplittedClasses));
40  helper def : elementIncludeBinding (param : ATL!
       SimpleOutPatternElement) : Boolean =
41    if not param.bindings.oclIsUndefined() then
42      param.bindings->select(e|e.value.oclIsTypeOf(ATL!
           NavigationOrAttributeCallExp))     ->select(e|e.
           value.name=self.getRemoveModelElementLeft).size()>0
43    else
44      false
45    endif;
46  helper def : simpleOutPatternElements : Sequence(ATL!
       SimpleOutPatternElement) =
47    self.deletedRule->first().outPattern.elements;
48  helper def : deleteRule (param : Sequence(ATL!MatchedRule))
        : Sequence(ATL!MatchedRule) =
49    let elements : Sequence(String) = self.getSplittedClasses
50    in elements->iterate(p; y : Sequence(ATL!MatchedRule) =
        param |
51      if self.contains(p, param) then
52        self.deleteRule(y->excluding(param->at(self.index(p,
             param))))
53      else
54        y
55      endif   )   ;
56  helper def : index (param1 : String, param2 : Sequence(ATL!
       MatchedRule)) : Integer =
57    param2->iterate(p; y : Integer = 0 |
58      if p.inPattern.elements->first().type.name = param1
           then
59        param2->indexOf(p)
60      else
61        y
62      endif
63    );
64
65  ----------RULES----------------
66  rule Module {
```

```
67    from
68      s : ATL!Module
69    to
70      t : ATL!Module (
71        name <- s.name,
72        libraries <- s.libraries,
73        isRefining <- s.isRefining,
74        inModels <- s.inModels,
75        outModels <- s.outModels,
76        elements <- self.deleteRule(s.elements)
77      )
78    do{
79      t.elements <- t.elements->append(thisModule.
            MatchedRule2MatchedRule(t.elements));
80      t.elements <- t.elements->append(thisModule.
            MatchedRule2MatchedRule2(t.elements));
81      } }
82  ------------ADD METACLASS---------------
83  lazy rule MatchedRule2MatchedRule {
84    from
85      s : ATL!MatchedRule
86    to
87      mr : ATL!MatchedRule (
88        name <- thisModule.getSplittedClasses->first(),
89        isAbstract <- false,
90        isRefining <- false,
91        inPattern <- ip_i_c2,
92        outPattern <- op_i_c2,
93        commentsBefore <- Set {'--SPLITTED RULE 1'}
94      ),
95      ip_i_c2 : ATL!InPattern (
96        elements <- Sequence{ipe_i_c2}
97      ),
98      ipe_i_c2 : ATL!SimpleInPatternElement (
99        varName <- self.deletedRule->first().inPattern.
              elements->first().varName,
100       type <- ipet_i_c2
101     ),
```

```
102     ipet_i_c2 : ATL!OclModelElement (
103       name <- thisModule.getSplittedClasses->first().concat
              ('_1'),
104       model <- om
105     ),
106     om : ATL!OclModel (
107       name <- self.sourceMM
108     ),
109     op_i_c2 : ATL!OutPattern (
110     )
111     do{
112       for (iterator in self.simpleOutPatternElements){
113         op_i_c2.elements <- op_i_c2.elements->append(
              thisModule.SOPE2SOPE(op_i_c2.elements));
114         self.index <- self.index + 1;
115       }
116     }
117 }
118 lazy rule SOPE2SOPE {
119   from
120     s : ATL!SimpleOutPatternElement
121   to
122     ope_i_c2 : ATL!SimpleOutPatternElement(
123       varName <- self.simpleOutPatternElements.at(self.
              index).varName,
124       type <- opet_i_c2
125     ),
126     opet_i_c2 : ATL!OclModelElement (
127       name <- self.simpleOutPatternElements.at(self.index).
              type.name,
128       model <- om2
129     ),
130     om2 : ATL!OclModel (
131       name <- self.targetMM
132     ),
133     b : ATL!Binding (
134       propertyName <- self.deletedBindings->first().
              propertyName,
```

```
135       value <- arg
136     ),
137     arg : ATL!NavigationOrAttributeCallExp (
138       name <- self.deletedBindings->first().value.name,
139       source <- ve
140     ),
141     ve : ATL!VariableExp (
142       referredVariable <- self.deletedBindings->first().
              value.source.referredVariable
143       )
144     do{
145       self.indexBinding <- 1;
146       for (iterator in self.simpleOutPatternElements.at(
              self.index).bindings){
147         if (self.simpleOutPatternElements.at(self.index).
              bindings.at(self.indexBinding).value.
              oclIsTypeOf(ATL!VariableExp)){
148           'B2B'.println();
149           ope_i_c2.bindings <- ope_i_c2.bindings->append(
                self.B2B(ope_i_c2.bindings));
150         }else{
151           if (self.simpleOutPatternElements.at(self.index).
                bindings.at(self.indexBinding).value.
                oclIsTypeOf(ATL!StringExp)){
152             'B2BString'.println();
153             ope_i_c2.bindings <- ope_i_c2.bindings->append(
                  self.B2BString(ope_i_c2.bindings));
154           }else{
155             'B2BNavigation'.println();
156             --We must filter bindings that are not present
157             --in the splitted class
158             if (not self.containsString(self.
                  simpleOutPatternElements.at(self.index).
                  bindings.at(self.indexBinding).value.name,
159                self.deletedAttributes(thisModule.
                     getSplittedClasses->first().concat('_1'))
                     )){              ope_i_c2.bindings <-
                     ope_i_c2.bindings->append(self.
```

```
                    B2BNavigation(ope_i_c2.bindings));
                          }
160           }
161         }
162         self.indexBinding <- self.indexBinding + 1;
163       }
164   }
165 }
166
167 --Binding for when binding.value is
168 --NavigationOrAttributeCallExp
169 lazy rule B2BNavigation {
170   from
171     s : ATL!Binding
172   to
173     b : ATL!Binding (
174       propertyName <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).
              propertyName,
175       value <- arg
176     ),
177     arg : ATL!NavigationOrAttributeCallExp (
178       name <- self.simpleOutPatternElements.at(self.index).
              bindings.at(self.indexBinding).value.name,
179       source <- ve
180     ),
181     ve : ATL!VariableExp (
182       referredVariable <- self.simpleOutPatternElements.at(
              self.index).bindings.at(self.indexBinding).value.
              source.referredVariable
183       )
184 }
185
186 --Binding for when binding.value is VariableExp
187 lazy rule B2B {
188   from
189     s : ATL!Binding
190   to
```

```
191     b : ATL!Binding (
192       propertyName <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).
              propertyName,
193       value <- ve
194     ),
195     ve : ATL!VariableExp (
196       referredVariable <- self.simpleOutPatternElements.at(
              self.index).bindings.at(self.indexBinding).value.
              referredVariable      )
197 }
198
199 --Binding for when binding.value is StringExp
200 lazy rule B2BString {
201   from
202     s : ATL!Binding
203   to
204     b : ATL!Binding (
205       propertyName <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).
              propertyName,
206       value <- se
207     ),
208     se : ATL!StringExp (
209       stringSymbol <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).value.
              stringSymbol      )
210 }
211
212 --------SPLITTED RULE 2-----------
213 lazy rule MatchedRule2MatchedRule2 {
214   from
215     s : ATL!MatchedRule
216   to
217     mr : ATL!MatchedRule (
218       name <- thisModule.getSplittedClasses->first().concat
              ('_2'),
219       isAbstract <- false,
```

```
220       isRefining <- false,
221       inPattern <- ip_i_c2,
222       outPattern <- op_i_c2,
223       commentsBefore <- Set {'--SPLITTED RULE 2'}
224     ),
225     ip_i_c2 : ATL!InPattern (
226       elements <- Sequence{ipe_i_c2}
227     ),
228     ipe_i_c2 : ATL!SimpleInPatternElement (
229       varName <- self.deletedRule->first().inPattern.
              elements->first().varName,
230       type <- ipet_i_c2
231     ),
232     ipet_i_c2 : ATL!OclModelElement (
233       name <- thisModule.getSplittedClasses->first().concat
              ('_2'),
234       model <- om
235     ),
236     om : ATL!OclModel (
237       name <- self.sourceMM
238     ),
239     op_i_c2 : ATL!OutPattern (
240
241     )
242   do{
243     self.index <- 1;
244       for (iterator in self.simpleOutPatternElements){
245         op_i_c2.elements <- op_i_c2.elements->append(
              thisModule.SOPE2SOPE2(op_i_c2.elements));
246         self.index <- self.index + 1;
247       }
248     }
249 }
250 lazy rule SOPE2SOPE2 {
251   from
252     s : ATL!SimpleOutPatternElement
253   to
254     ope_i_c2 : ATL!SimpleOutPatternElement(
```

```
255       varName <- self.simpleOutPatternElements.at(self.
              index).varName,
256       type <- opet_i_c2
257     ),
258     opet_i_c2 : ATL!OclModelElement (
259       name <- self.simpleOutPatternElements.at(self.index).
              type.name,
260       model <- om2
261     ),
262     om2 : ATL!OclModel (
263       name <- self.targetMM
264     ),
265     b : ATL!Binding (
266       propertyName <- self.deletedBindings->first().
              propertyName,
267       value <- arg
268     ),
269     arg : ATL!NavigationOrAttributeCallExp (
270       name <- self.deletedBindings->first().value.name,
271       source <- ve
272     ),
273     ve : ATL!VariableExp (
274       referredVariable <- self.deletedBindings->first().
              value.source.referredVariable
275     )
276     do{
277       self.indexBinding <- 1;
278       for (iterator in self.simpleOutPatternElements.at(
              self.index).bindings){
279         if (self.simpleOutPatternElements.at(self.index).
              bindings.at(self.indexBinding).value.
              oclIsTypeOf(ATL!VariableExp)){
280           'B2B'.println();
281           ope_i_c2.bindings <- ope_i_c2.bindings->append(
              self.B2B2(ope_i_c2.bindings));
282         }else{
283           if (self.simpleOutPatternElements.at(self.index).
              bindings.at(self.indexBinding).value.
```

```
              oclIsTypeOf(ATL!StringExp)){
284           'B2BString'.println();
285           ope_i_c2.bindings <- ope_i_c2.bindings->append(
                 self.B2BString2(ope_i_c2.bindings));
286         }else{
287           'B2BNavigation'.println();
288           --We must filter bindings that are not present
289           --in the splitted class
290           if (not self.containsString(self.
                 simpleOutPatternElements.at(self.index).
                 bindings.at(self.indexBinding).value.name,
291               self.deletedAttributes(thisModule.
                    getSplittedClasses->first().concat('_2'))
                    )){
292               ope_i_c2.bindings <- ope_i_c2.bindings->
                    append(self.B2BNavigation2(ope_i_c2.
                    bindings));
293           }
294         }
295       }
296       self.indexBinding <- self.indexBinding + 1;
297     }
298   }
299 }
300 --Binding for when binding.value is
301 --NavigationOrAttributeCallExp
302 lazy rule B2BNavigation2 {
303   from
304     s : ATL!Binding
305   to
306     b : ATL!Binding (
307       propertyName <- self.simpleOutPatternElements.at(self
             .index).bindings.at(self.indexBinding).
             propertyName,       value <- arg
308     ),
309     arg : ATL!NavigationOrAttributeCallExp (
310       name <- self.simpleOutPatternElements.at(self.index).
             bindings.at(self.indexBinding).value.name,
```

```
311       source <- ve
312     ),
313     ve : ATL!VariableExp (
314       referredVariable <- self.simpleOutPatternElements.at(
              self.index).bindings.at(self.indexBinding).value.
              source.referredVariable
315     )
316 }
317 --Binding for when binding.value is VariableExp
318 lazy rule B2B2 {
319   from    s : ATL!Binding
320   to
321     b : ATL!Binding (
322       propertyName <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).
              propertyName,
323       value <- ve
324     ),
325     ve : ATL!VariableExp (
326       referredVariable <- self.simpleOutPatternElements.at(
              self.index).bindings.at(self.indexBinding).value.
              referredVariable
327     )
328 }
329 --Binding for when binding.value is StringExp
330 lazy rule B2BString2 {
331   from
332     s : ATL!Binding
333   to
334     b : ATL!Binding (
335       propertyName <- self.simpleOutPatternElements.at(self
              .index).bindings.at(self.indexBinding).
              propertyName,
336       value <- se
337       --commentsBefore <- Set {'--comment'}
338     ),
339     se : ATL!StringExp (
```

```
340         stringSymbol <- self.simpleOutPatternElements.at(self
                .index).bindings.at(self.indexBinding).value.
                stringSymbol
341         )
342       do{
343       }
344 }
345 ---------ADD METAPROPERTY------------
346 rule SimpleOutPatternElement1 {
347   from
348     s : ATL!SimpleOutPatternElement
349   to
350     t : ATL!SimpleOutPatternElement(
351       type <- s.type,
352       varName <- s.varName,
353       bindings <- s.bindings
354     )
355 }
```

Listing A.1: Split class adaptation

Appendix B

M2T Transformation Adaptation

This appendix shows one of the adaptations presented in Chapter 5 as an example. Specifically, *remove column* case is shown.

```
1  diff.objectsOfType(diff.RemoveModelElement)->forEach(rme:
       diff.RemoveModelElement | rme.rightParent != "" and rme
       .rightParent=tableName and tableName != ""){
2    stdout.println("REMOVE COLUMN");
3    removeColumn = true;
4    var paramsRemoveColumn:List;
5    paramsRemoveColumn.add(statement);
6    paramsRemoveColumn.add(rme.rightParent);
7    paramsRemoveColumn.add(rme.leftElement.substringAfter("/"
         ));
8    println("#ADAPTED: Removed column "+rme.leftElement.
         substringAfter("/") +" from table: " +tableName);
9    println("#"+statement);
10   println(java("org.gibello.zql.ZqlParser", "removeColumn",
         paramsRemoveColumn,CLASSPATH )+";");
11   }
```

Listing B.1: M2T transformation adaptation

```
1  /*
2    * Depending on the corresponding first word: select,
3    *insert, update or delete
4    * call the corresponding function:
```

```
5     *removeColumnSelect, ...
6   */
7     public static String removeColumn(String pStatement,
          String pTable, String pColumn) {
8       if(pStatement.toLowerCase().startsWith("select")){
9         return removeColumnSelect(pStatement, pTable,
              pColumn);
10        }else if (pStatement.toLowerCase().startsWith("
              insert")){
11          return removeColumnInsert(pStatement, pTable,
                pColumn);
12        }else if (pStatement.toLowerCase().startsWith("
              update")){
13          return removeColumnUpdate(pStatement, pTable,
                pColumn);
14        }else if (pStatement.toLowerCase().startsWith("
              delete")){
15        return removeColumnDelete(pStatement, pTable,
              pColumn);
16        }else{
17        return "";
18       }
19      }
20
21  public static String removeColumnInsert(String pStatement,
        String pTable, String pColumn) {
22      InputStream stream = null;
23      try {
24        stream = new ByteArrayInputStream(pStatement.getBytes
              ("UTF-8"));
25      } catch (UnsupportedEncodingException e) {
26        e.printStackTrace();
27      }
28        ZqlParser p = new ZqlParser(stream);
29        ZInsert zq = null;
30      try {
31        zq = (ZInsert)p.readStatement();
32      }catch (ParseException e) {
```

```
33      e.printStackTrace();
34    }
35    if(zq.getTable().compareTo(pTable)==0){
36      //DELETE COLUMN
37        Vector vector = zq.getColumns();
38        //iterate columns
39        Iterator it = vector.iterator();
40        int i = 0;
41        int affectedPosition = 0;
42        while(it.hasNext()){
43          String column = (String)it.next();
44          //if we find the column, delete it
45          if(column.compareTo(pColumn) == 0){
46            affectedPosition = i;
47          }
48          i++;
49        }
50        vector.remove(affectedPosition);
51        zq.addColumns(vector);
52        //DELETE VALUE
53        vector = zq.getValues();
54        vector.removeElementAt(affectedPosition);
55    }
56        return zq.toString();
57  }
58
59  public static String removeColumnSelect(String pStatement,
        String pTable, String pColumn) {
60    InputStream stream = null;
61    try {
62      stream = new ByteArrayInputStream(pStatement.getBytes
          ("UTF-8"));
63    } catch (UnsupportedEncodingException e) {
64      e.printStackTrace();
65    }
66      ZqlParser p = new ZqlParser(stream);
67      ZQuery zq = null;
68    try {
```

```
69        zq = (ZQuery)p.readStatement();
70      } catch (ParseException e) {
71        e.printStackTrace();
72      }
73      Vector vector = zq.getFrom();
74        //iterate tables of statement to find table name
75        Iterator it = vector.iterator();
76        boolean isTable = false;
77        while(it.hasNext()){
78          ZFromItem zfi = (ZFromItem)it.next();
79          //check if it is the table
80          if(zfi.getTable().compareTo(pTable) == 0){
81            isTable = true;
82          }
83        }
84        //if it is the table, look for the column and
85      //rename it
86        if(isTable){
87           Vector vector2 = zq.getSelect();
88           Iterator it2 = vector2.iterator();
89           int i = 0;
90           int affectedPosition = 0;
91           while(it2.hasNext()){
92             ZSelectItem select = (ZSelectItem)it2.next();
93             if(select.getColumn().compareTo(pColumn)==0){
94               affectedPosition = i;
95             }
96             i++;
97           }
98           vector2.remove(affectedPosition);
99        }
100         return zq.toString();
101   }
102
103 public static String removeColumnUpdate(String pStatement,
      String pTable, String pColumn) {
104     InputStream stream = null;
105     try {
```

```
106       stream = new ByteArrayInputStream(pStatement.getBytes
              ("UTF-8"));
107     } catch (UnsupportedEncodingException e) {
108       e.printStackTrace();
109     }
110       ZqlParser p = new ZqlParser(stream);
111       ZUpdate zq = null;
112     try {
113       zq = (ZUpdate)p.readStatement();
114     }catch (ParseException e) {
115       e.printStackTrace();
116     }
117     if(zq.getTable().compareTo(pTable)==0){
118       //remove column of SET part
119       Hashtable hash = (Hashtable)zq.getSet();//key:column
120       Enumeration enume = hash.keys();
121       String column = "";
122       boolean affectedSet = false;
123       while(enume.hasMoreElements()){
124         column = (String)enume.nextElement();
125         if(column.compareTo(pColumn)==0){
126           affectedSet = true;
127         }
128       }
129       if(affectedSet){
130         zq.removeColumnUpdate(pColumn);
131       }
132       //remove column from WHERE part
133        ZExpression ze = (ZExpression)zq.getWhere();
134        DeleteZExpressionFromZExpression(pColumn, ze);
135     }
136      return zq.toString();
137  }
138
139 public static String removeColumnDelete(String pStatement,
       String pTable, String pColumn) {
140     InputStream stream = null;
141     try {
```

```
142       stream = new ByteArrayInputStream(pStatement.getBytes
              ("UTF-8"));
143     } catch (UnsupportedEncodingException e) {
144       e.printStackTrace();
145     }
146       ZqlParser p = new ZqlParser(stream);
147       ZDelete zq = null;
148     try {
149       zq = (ZDelete)p.readStatement();
150     } catch (ParseException e) {
151       e.printStackTrace();
152     }
153     if(zq.getTable().compareTo(pTable)==0){
154       //remove column from WHERE part
155        ZExpression ze = (ZExpression)zq.getWhere();
156        DeleteZExpressionFromZExpression(pColumn, ze);
157     }
158      return zq.toString();
159 }
```

Listing B.2: Auxiliar Java code to modify SQL statements

Bibliography

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of Model Transformations via Alloy. In Benoit Baudry, Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Proceedings of the 4th workshop Model-Driven Engineering, Verification and Validation (MoDeVVa 2007), Nashville, TN, USA*, pages 47–56, 2007.

[AC07] Michal Antkiewicz and Krzysztof Czarnecki. Design Space of Heterogeneous Synchronization. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Braga, Portugal*, volume 5235 of *Lecture Notes in Computer Science*, pages 3–46. Springer, 2007.

[And94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. Technical Report AIB 94-12, RWTH Aachen, 1994.

[BÓ4] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2):21–24, 2004.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, and Arne Lindow. Model

Transformations? Transformation Models! In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy*, 2006.

[BBJ+10] Gabriel Barbier, Hugo Bruneliere, Frédéric Jouault, Yves Lennon, and Frédéric Madiot. MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases. In William M. Ulrich and Philip H. Newcomb, editors, *Information Systems Transformation*, The MK/OMG Press, pages 365 – 400. Morgan Kaufmann, Boston, 2010.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[BDTM+06] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model Transformation Testing Challenges. In *Proceedings of the IMDDMDT workshop at European Conference on Model Driven Architecture (ECMDA 2006)*, Bilbao, Spain, July 2006.

[Bez90] Boris Bezier. *Software Testing Techniques*. New York : Van Nostrand Reinhold, 1990.

[BFS+06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006),Raleigh, USA*, 2006.

[BGF+10] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6):139–143, 2010.

[Bos09] Jan Bosch. From Software Product Lines to Software Ecosystems. In Dirk Muthig and John D. McGregor, editors, *13th International Software Product Line Conference (SPLC 2009), San Francisco, California, USA*, volume 446 of *ACM International Conference Proceeding Series*, pages 111–119. ACM, 2009.

[Bro87] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSyM)*, 4(2):171–188, 2005.

[CBH10] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A Conceptual Approach to Database Applications Evolution. In *International Conference on Conceptual Modeling (ER 2010), Vancouver, Canada*, pages 132–145, 2010.

[CBS12] Juan José Cadavid, Benoit Baudry, and Houari A. Sahraoui. Searching the Boundaries of a Modeling Space to Test Metamodels. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *International Conference on Software Testing (ICST 2012), Montreal, Quebec, Canada*, pages 131–140. IEEE, 2012.

[CC04] Jordi Cabot and Jordi Conesa. Automatic Integrity Constraint Evolution due to Model Subtract Operations.

In Shan Wang, Dongqing Yang, Katsumi Tanaka, Fabio Grandi, Shuigeng Zhou, Eleni E. Mangina, Tok Wang Ling, Il-Yeol Song, Jihong Guan, and Heinrich C. Mayr, editors, *23rd International Conference on Conceptual Modeling (Workshops) (ER 2004), Shanghai, China*, volume 3289 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2004.

[CH05] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in Database Applications Evolution. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Braga, Portugal*, pages 409–421, 2005.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[CIGM14] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting Models from Source Code in Software Modernization. *Software & Systems Modeling (SoSyM)*, 13(2):713–734, 2014.

[CJCGM11] Javier Luis Canovas, Frederic Jouault, Jordi Cabot, and Jesús Garcia Molina. API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering. *Information and Software Technology*, 54(3):257–273, October 2011.

[CMDZ13] Carlos Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. *Very Large Databases (VLDB 2013), Trento, Italy*, 22(1):73–98, 2013.

[CMT06] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez Tortosa. RubyTL: A Practical, Extensible Transformation Language. In Arend Rensink and Jos Warmer, editors, *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain*, volume 4066 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2006.

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[CRIP13] Antonio Cicchetti, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Managing the Evolution of Data-intensive Web Applications by Model-driven Techniques. *Software and System Modeling (SoSyM)*, 12(1):53–83, 2013.

[CTMZ08] Carlos Curino, Letizia Tanca, Hyun Jin Moon, and Carlo Zaniolo. Schema Evolution in Wikipedia: Toward a Web Information System Benchmark. In *10th International Conference on Enterprise Information Systems (ICEIS 2008), Barcelona, Spain*, 2008.

[DGB12] Gaëtan Deltombe, Olivier Le Goaer, and Franck Barbier. Bridging KDM and ASTM for Model-Driven Software Modernization. In *24th International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), San Francisco Bay, USA*, pages 517–524. Knowledge Systems Institute Graduate School, 2012.

[DJ06] Danny Dig and Ralph Johnson. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance*, 18(2):83–107, 2006.

[DLRZ08] Eladio Domínguez, Jorge Lloret, Angel L. Rubio, and Maria A. Zapata. MeDEA: A Database Evolution Architecture with Traceability. *Data Knowledge Engineering*, 65(3):419–441, 2008.

[DRIP11] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. What is Needed for Managing Co-evolution in MDE? In *Proc. of the 2nd International Workshop on Model Comparison in Practice (IWMCP 2011), Esslingen, Germany*, pages 30–38, New York, NY, USA, 2011. ACM.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*. Eclipse, 2006.

[FBMT09] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying Input Test Data for Model Transformations. *Software and System Modeling (SoSyM)*, 8(2):185–203, 2009.

[FS03] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in Model-driven Engineering: Testing Model Transformations. In *Model, Design and Validation (MoDeVa 2004)*, pages 29–40, Nov 2004.

[FV04] Lidia Fuentes and Antonio Vallecillo. An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13, 2004.

[GC12] Carlos A. González and Jordi Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In *15th International Conference Model Driven Engineering Languages and Systems (MODELS 2012), Innsbruck, Austria*, 2012.

[GJCB08] Kelly Garces, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Adaptation of Models to Evolving Metamodels. Rapport de recherche RR-6723, INRIA, 2008.

[GJCB09a] Kelly Garcés, Frederic Jouault, Pierre Cointe, and Jean Bézivin. A Domain Specific Language for Expressing Model Matching. In *Proc. of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09)*, 2009.

[GJCB09b] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009), Enschede, The Netherlands*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.

[Gue12] Esther Guerra. Specification-Driven Test Generation for Model Transformations. In Zhenjiang Hu and Juan de Lara, editors, *International Conference on Model Transformations (ICMT 2012), Prague, Czech Republic*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2012.

[GV11] Martin Gogolla and Antonio Vallecillo. Tractable Model Transformation Testing. In Robert France, Jochen Kuester, Behzad Bordbar, and Richard Paige, editors, *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 221–235. Springer Berlin Heidelberg, 2011.

[Her11] Markus Herrmannsdoerfer. COPE A Workbench for the Coupled Evolution of Metamodels and Models. In *Software Language Engineering (SLE 2011), Braga, Portugal*. Springer Berlin / Heidelberg, 2011.

[HH06] Jean-Marc Hick and Jean-Luc Hainaut. Database Application Evolution: A Transformational Approach. *Data Knowledge Engineering*, 59(3):534–558, 2006.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the Gap between Modelling and Java. In *Software Language Engineering (SLE 2009), Denver, CO, USA*. Springer, 2009.

[HRW09] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language Evolution in Practice: The History of GMF. In *Software Language Engineering (SLE 2009), Denver, CO, USA*, 2009.

[HRW11] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 633–642, New York, NY, USA, 2011. ACM.

[HVW11] Markus Herrmannsdoefer, Sander Vermolen, and Guido Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In

3rd International Conference on Software Language Engineering (SLE 2010), Eindhoven, The Netherlands, October 12-13, 2010, 2011.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *33rd International Conference on Software Engineering (ICSE 2011), Waikiki, Honolulu, Hawaii*, pages 471–480. ACM, 2011.

[IEE99] IEEE. *IEEE Standard for Software Maintenance, IEEE Std 1219-1998*, volume 2. IEEE Press, 1999.

[IPM12] Ludovico Iovino, Alfonso Pierantonio, and Ivano Malavolta. On the Impact Significance of Metamodel Evolution in MDE. *Journal of Object Technology (JOT)*, 11(3):3: 1–33, 2012.

[ISO01] ISO. International Standard ISO/IEC 9126, Information Technology Product Quality Part1: Quality Model. Technical report, International Standard Organization, 2001.

[JABK08] Frederic Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming (SCP)*, 72(1-2):31–39, 2008.

[JAOA13] J.V. Joshua, D.O. Alao, S.O. Okolie, and O. Awodele. Software Ecosystem: Features, Benefits and Challenges. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 4(8), 2013.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete

Syntaxes in Model Engineering. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *5th International Conference Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA*, pages 249–254. ACM, 2006.

[KAER06] Jochen M. Kuster and Mohamed Abd-El-Razik. Validation of Model Transformations - First Experiences using a White Box Approach. In *3rd International Workshop on Model Development, Validation and Verification (MoDeVa 2006), Genova, Italy*, 2006.

[Ken02] Stuart Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.

[KKBG13] Jochen Küster, Daniel Kovacs, Eduard Bauer, and Christian Gerth. Integrating Coverage Analysis into Test-driven Development of Model Transformations. IBM Research Report RZ 3846, IBM Research - Zurich, April 2013.

[Kla08] Benjamin Klatt. Xpand: A Closer Look at the Model2text Transformation Language, 2008.

[KR03] Vinay Kulkarni and Sreedhar Reddy. Separation of Concerns in Model-Driven Development. *IEEE Softw.*, 20(5):64–69, September 2003.

[Kur05] Ivan Kurtev. *Adaptability of model transformations*. PhD thesis, Enschede, the Netherlands, May 2005.

[LBNK10] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A Novel Approach to Semi-automated Evolution of DSML Model

Transformation. In *Software Language Engineering (SLE 2010), Eindhoven, The Netherlands*. Springer Berlin / Heidelberg, 2010.

[LWK10] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-Model Transformations By Demonstration. In Laurence Tratt and Martin Gogolla, editors, *International Conference on Model Transformation (ICMT 2010), Malaga, Spain*, volume 6142 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2010.

[LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray. *A Testing Framework for Model Transformations*, chapter 10, pages 219–236. Model-Driven Software Development - Research and Practice in Software Engineering, Springer, 2005.

[MCD+08] Hyun Jin Moon, Carlo Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and Querying Transaction-time Databases under Schema Evolution. *Proc. VLDB*, 1(1):882–895, 2008.

[MD08] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA 2008), Berlin, Germany*, pages 432–443. Springer-Verlag, 2008.

[Men08] Tom Mens. Introduction and Roadmap: History and Challenges of Software Evolution. In Tom Mens and Serge Demeyer, editors, *Software Evolution*, pages 1–11. Springer, 2008.

[MFB09] Pierre-Alain Muller, Frédéric Fondement, and Benoit Baudry. Modeling Modeling. In Andy Schürr and Bran Selic, editors, *12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado, USA*, volume 5795 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2009.

[MM03] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[Mod03] Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach, june 2003.

[MP09] Jacqueline A. McQuillan and James F. Power. White-Box Coverage Criteria for Model Transformations. In *1st International Workshop on Model Transformation with ATL (MtATL 2009), Nantes, France*, 2009.

[MSTC12] Jean-Marie Mottu, Sagar Sen, Massimo Tisi, and Jordi Cabot. Static Analysis of Model Transformations for Effective Test Generation. In *ISSRE - 23rd IEEE International Symposium on Software Reliability Engineering*, Dallas, États-Unis, 2012.

[OH07] Jon Oldevik and Øystein Haugen. Higher-Order Transformations for Product Lines. In *6th International Software Product Line Conference (SPLC 2007), Kyoto, Japan*, pages 243–254, 2007.

[OMG] OMG. Unified Modeling Language http://www.omg.org/spec/UML/2.2/Superstructure/PDF/.

[OMG02] OMG. Meta Object Facility (MOF) Specification - Version 1.4. Adopted Specification http://www.omg.org/spec/MOF/1.4/PDF/, April 2002.

[OMG11a] OMG. Architecture-Driven Modernization: Abstract Syntax Tree Metamodel (ASTM), http://www.omg.org/spec/ASTM/1.0/, 1 2011.

[OMG11b] OMG. Architecture-Driven Modernization (ADM)/ Knowledge Discovery Meta-Model (KDM), http://www.omg.org/spec/KDM/1.3/, August 2011.

[OMG11c] OMG. Query/View/Transformation, v1.1. Formal Specification, January 2011. Online at: http://www.omg.org/spec/QVT/1.1/PDF/.

[OO07] Gøran K. Olsen and Jon Oldevik. Scenarios of Traceability in Model to Text Transformations. In *3rd European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, 2007.

[PD99] Norman W. Paton and Oscar Díaz. Active Database Systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[PD12] Gorka Puente and Oscar Díaz. Wiki Refactoring as Mind Map Reshaping. In *International Conference on Advanced Information Systems Engineering (CAiSE 2012), Gdansk, Poland*, pages 646–661, 2012.

[PDA13] Gorka Puente, Oscar Díaz, and Maider Azanza. Refactoring Affordances in Corporate Wikis: a Case for the Use of Mind Maps. *Enterprise Information Systems*, 0(0):1–50, 2013.

[Ra04] Young-Gook Ra. Relational Schema Evolution for Program Independency. In *Proceedings of the 7th International Conference on Intelligent Information Technology (CIT 2004), Hyderabad, India*, pages 273–281. Springer-Verlag, 2004.

[RIP12a] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Coupled Evolution in Model-Driven Engineering. *IEEE Software*, 29(6):78–84, 2012.

[RIP12b] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *6th International Conference on Graph Transformation (ICGT 2012), Bremen, Germany*, volume 7562 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2012.

[RIP13] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Managing the Coupled Evolution of Metamodels and Textual Concrete Syntax Specifications. In Onur Demirörs and Oktay Türetken, editors, *EUROMICRO-SEAA*, pages 114–121. IEEE, 2013.

[RKPP10] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In *International Conference on Model Transformation (ICMT 2010), Malaga, Spain*, pages 184–198, 2010.

[RLP10] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated Co-evolution of GMF Editor Models. *CoRR*, abs/1006.5761, 2010.

[RLR98] Elke A. Rundensteiner, Amy J. Lee, and Young-Gook Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. In *In Object-Oriented Information Systems*. Springer, 1998.

[RPKP08] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. In Ina Schieferdecker and Alan Hartman, editors, *4th European Conference on Model Driven Architecture - Foundation and Applications (ECMDA-FA 2008), Berlin, Germany*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[SBM08] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *1st International Conference on Software Testing, Verification, and Validation (ICST 2008), Lillehammer, Norway*, 2008.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2 edition, 2009.

[SCD12] Gehan M. K. Selim, James R. Cordy, and Juergen Dingel. Model Transformation Testing: The State of the Art. In *1st Workshop on the Analysis of Model Transformations (AMT 2012), Innsbruck, Austria*, 2012.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SDP+10] B. Schätz, D. Deridder, A. Pierantonio, J. Sprinkle, and D. Tamzalit. On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In *Proc. of the International Workshop on Models and Evolution (ME 2011) at Model Driven Engineering Languages and Systems (MoDELS 2011), Wellington, New Zealand*, pages 54–63, 2010.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SG13] Yu Sun and Jeff Gray. End-User Support for Debugging Demonstration-Based Model Transformation Execution. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013), Montpellier, France*, 2013.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On Model Typing. *Software and System Modeling (SoSyM)*, 6(4):401–413, 2007.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: the Heart and Soul of Model-driven Software Development. *Software, IEEE*, 20(5):42–45, 2003.

[Sne00] Harry M. Sneed. Encapsulation of Legacy Software: A Technique for Reusing Legacy Software Components. *Annals of Software Engineering*, 9(1-4):293–313, 2000.

[THHB06] Philippe Thiran, Jean-Luc Hainaut, Geert-Jan Houben, and Djamal Benslimane. Wrapper-based Evolution of Legacy Information Systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, October 2006.

[TJF+09] Massimo Tisi, Frederic Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *European Conference On Model Driven Architecture: Foundations and Applications (ECMDA-FA 2009), Enschede, The Netherlands*, 2009.

[Tou06] Antoine Toulmé. Presentation of EMF Compare Utility. *Eclipse Modeling Symposium 2006*, pages 1–8, 2006.

[TRL12] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. Early Experiences on Model Transformation Testing. In *1st Workshop on the Analysis of Model Transformations (AMT 2012), Innsbruck, Austria*, 2012.

[vAvdB11] Marcel van Amstel and Mark G. J. van den Brand. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In *International Conference on Model Transformation (ICMT 2011), Zurich, Switzerland*, pages 108–122, 2011.

[vdBPV11] Mark van den Brand, Zvezdan Protic, and Tom Verhoeff. A Generic Solution for Syntax-Driven Model Co-evolution. In Judith Bishop and Antonio Vallecillo, editors, *49th International Conference on Objects, Models, Components and Patterns (TOOLS 2011), Zurich - Switzerland*, volume 6705 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2011.

[VMP04] Yannis Velegrakis, Renee J. Miller, and Lucian Popa. Preserving Mapping Consistency under Schema Changes. *The VLDB Journal*, 13(3):274–293, September 2004.

[VWV12] Sander Vermolen, Guido Wachsmuth, and Elco Visser. Reconstructing Complex Metamodel Evolution. In Uwe Assmann and Tony Sloane, editors, *Software Language Engineering, Fourth International Conference, SLE 2011, Braga, Portugal, July, 2011, Revised Selected Papers*. Springer, 2012.

[VWVDVD07] Eelco Visser, Jos Warmer, Arie Van Deursen, and Arie Van Deursen. Model-driven software evolution: A research agenda. In *In Proc. Int. Ws on Model-Driven Software Evolution held with the ECSMR'07*, 2007.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP 2007)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.

[WB13] Manuel Wimmer and Loli Burgueño. Testing M2T/T2M Transformations. In *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2013), Miami, USA*, 2013.

[WKC06] Junhua Wang, Soon-Kyeong Kim, and David A. Carrington. Verifying Metamodel Coverage of Model Transformations. In *Australasian Software Engineering Conference (ASWEC 2006)*, pages 270–282. IEEE Computer Society, 2006.

[WKK+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In Laurence Tratt and Martin Gogolla, editors, *International Conference on Model Transformation (ICMT 2010), Malaga, Spain*, volume 6142 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2010.

Acknowledgments

Developing a Thesis, is not a one-person task (at least for ordinary people). Many persons and institutions are directly and indirectly involved, and it is a matter of justice to acknowledge their support.

First and foremost, the help of my supervisor Professor Oscar Díaz has been invaluable. Although his tireless advocacy of the importance of the problem statement, again and again I run into the same stone. In this sense, his patience has been endless. Although my talent and knowledge always will be very far from him, it has been and will be an inspiring example that will make me keeping on.

A favorable environment has ease the development of this Thesis. Working on a research group collaborating with other members has been enriching. I would like to express my gratitude to my colleagues along this stage: Jon Iturrioz, Arantza Irastorza, Maider Azanza, Felipe Ibañez, Felipe Martin, Luis Mari Alonso, Iker Azpeitia, Iñaki Paz, Sandy Pérez, Cristóbal Arellano, Gorka Puente, Itziar Otaduy, Iñigo Aldalur and Leticia Montalvillo ;)

I also want to thank Jordi Cabot, leader of Atlanmod group, for admitting me in a research visit. Of course, I don't forget other members of the group: Javier Cánovas, Carlos Gonzalez, Hugo Bruneliere, Salva Martinez, Fabian Buttner, Massimo Tisi, Sagar Sen and Valerio Cosetino. They make me feel like at home and I had a productive and very funny time there.

And last but not least, this Thesis has been feasible thanks to the economical support of the Basque Government's pre-doctoral grant under

the “Researchers Training Program” during the years 2010 to 2013. It has been also supported by the *Spanish Ministry of Science and Education*.