Metamodelisation to support Test and Evolution

Anne Etien

Université Lille 1, Sciences et Technologies

Spécialité Informatique — École doctorale Science Pour l'Ingénieur (Lille)

Metamodelisation to support Test and Evolution

Habilitation à Diriger les Recherches

Université Lille 1, Sciences et Technologies

(spécialité informatique)

présentée et soutenue publiquement le 28 juin 2016

par

Anne Etien

Composition du jury

Président :	Mme Laurence Duchien
Rapporteurs :	M. Serge Demeyer, Mme Marianne Huchard, M. Jurgen Vinju
Examinateur :	M. Xavier Blanc
Garant :	M. Stéphane Ducasse

Centre de Recherche en Informatique Signal et Automatique de Lille — UMR CNRS 9189
INRIA Lille - Nord Europe

Numéro d'ordre: XXXXX

Contents

CHAPTER 1

Introduction

During ten years, I have been working on two different domains *Model Driven Engineering* and *Software Maintenance* but with a single target: *Designing systems of good quality, easily maintainable*. The reasoning was also the same: I proposed solutions based on metamodels or models to provide genericity and independence from the programming languages of the results. The considered systems are either chains of model transformations (*i.e.* not the program generated from models but the, model based, compiler itself) or traditional programs. Quality can be ensured and measured from different ways. In this document, I only focus on tests.

1.1 Goal of this Document

In this document of Habilitation to Supervise Research, I aimed at illustrating three complementary qualities that I consider necessary to supervise research:

- *ability to supervise novice or young researchers*. Our mission as supervisor is to teach novice and young researchers to search without mandatorily obtaining results, to search again and find relevant results and to present them. I had the opportunity to supervise 4 PhD students (including one who already defended) and 4 master students. I also accompanied in their research, one post doc and one ATER.

- *ability to collaborate*. In my opinion, research is synonym of exchange. Exchange between people from different horizons with different backgrounds and ways of thinking. Collaborations can be either academic, between researchers, or industrial to answer real problems. I worked in close collaboration with French or international researchers. Moreover, since I had the opportunity to do my own research, I have been trying to answer concrete and real issues. Some were initiated by companies, others were transferred to the industry.

- *ability to have a vision*. Research in a domain does not stop with the defense of a PhD student, or the end of a project. On the contrary, it is mostly the occasion to raise new issues. Having a vision, is also having the ability to decompose long term research topics into short or mid term issues.

All the results presented in this document were reached in collaboration with either a novice researcher that I supervised or a colleague. Details of the presented results can be found in articles published in international journals, conferences and workshops.

1.2 Software Maintenance, Testing and Evolution

This document presents my work of these ten last years. My career follows a "traditional" path. After a PhD at the University Paris 1 on information systems, I came to Lille in 2006 for a postdoctoral stay to work on model transformations in the Dart team common to Inria and Lifl Lab (now CRIStAL). After one year, I was enrolled as associate professor teaching at Polytech Lille and doing my research in the Dart team. Its core business was real-time embedded systems dealing with massively parallel data. The embedded systems were generated from models using model transformation chains. My research concerned the chains. In 2011, the Inria team stopped. A new one was created without any reference to models. My work had no place in this new organisation. After trying during one year to work alone or move to somewhere else, I finally decided to join the RMod team common to Inria and Lifl lab. The goal of RMoD is to support remodularisation and development of modular object-oriented applications. This objective is tackled from two complementary perspectives: reengineering and constructs for dynamic programming languages. I have been working on the reengineering part where we propose new analyses to understand and restructure existing large applications based on abstract representations.

Looking back to these years, two topics are constantly studied: testing and evolution. The analysed systems are different, model transformation chains or traditional programs but the main objective remains the same: ease maintenance. It can be noticed that this thematic was already strongly present in my PhD dissertation. Moreover, the used means *i.e.,* modelling and metamodelling also federates my research.

Before briefly presenting the topics tackled in this document, I explain why I am studying software maintenance.

Why Software Maintenance? Several studies showed that activities after delivery are pre-dominant in software engineering and correspond to 90% of the total cost of a typical software [Pigoski 1997, Seacord 2003]. These activities correspond to *software maintenance* that is the modification of a software product after delivery to correct faults, to improve performance or other attributes [ISO 2006]. Past studies (reported in [Pigoski 1997]) showed that, contrary to common belief, *corrective* maintenance (*i.e.* diagnosing and fixing errors) represents only a small part of all maintenance (21%). Most software maintenance (50%) is done to add new features (*perfective* maintenance). The modification of the system to cope with changes in the software environment (*adaptive* maintenance) corresponds to 25%. The system is modified to increase its maintainability or reliability (4%) to prevent problems in the future (*preventive* maintenance).

Moreover, Lehman's first law of software evolution (law of Continuing Change, [Lehman 1980]) specifies that "*a program that is used undergoes continual change or becomes progressively less useful*." A corollary to this law is that software maintenance is a sign of success: considering the costs associated to software maintenance, it is performed only for software systems which utility is perceived as more valuable than this cost. This is a conclusion that goes against the usual perception of the activity, but maintenance actually means that a system is useful and that its users see a value in its continuing operation.

Anticipation and Architecture Modifications. Object, aspect and model paradigms were introduced to enhance reusability, modularity and ease successive evolutions the software systems meet. Even if these paradigms were introduced for these purposes, for each software system, maintenance must be anticipated from the design phase [Budgen 2003]. However, everything cannot be anticipated and anyway, according to Lehman's second law (law of Increasing Complexity, [Lehman 1980]) "*as a program evolves, its complexity increases unless work is done to maintain or reduce it*". Modifications of the software architecture is required to reduce complexity and to bring new and solid bases for future evolutions. Support in the form of concepts, methods, techniques, and tools for recognizing, confronting, and managing architecture modifications is required [Avgeriou 2013].

Test to Ensure Quality and Ease Maintenance. According to Beck, the most important rule of simple design is "Passes the tests" [Beck 2004]. The point is that whatever else is done with a software system, the primary aim is that it works as intended and tests are there to verify that this happens.

Depending on their nature, tests answer different purposes. For example, they enable the developer to identify and locate errors in the code that thus, will not occur later. Moreover, tests ensure non regression after an evolution by checking that what was changed did not impact the rest of the system. For these two reasons, presence of tests positively acts on maintenance.

However, covering the whole code and all the alternatives with tests may not be possible in the context of large systems. Consequently, test sets have to be qualified to be considered good enough to highlight errors. On the other hand, when tests are numerous and code evolves it can be very long to execute all the tests after a change. For this reason, and also because unfortunately tests are often considered as lost time, big companies may not always put as much emphasis on tests as they should. Errors are thus detected only after delivery.

Software Ecosystem. Software artefacts are not independent anymore, they constitute ecosystems where the evolution of one element impacts the others. Consequently, evolution has to be thought in terms of co-evolution, the evolution of different artifacts in parallel or in response to a first change.

1.3 Models Everywhere

Model Driven Engineering. Model driven engineering and model transformation were introduced in the early 2000s with the goal to develop once and generate several times. If the theory was attractive, model transformation applications on real cases were chaotic. Indeed, model transformations were not designed to be maintained later. This new paradigm, model as first class artefacts, required to adapt existing technologies in term of design, test, maintenance and evolution.

This document tackles the issues of evolution, test and maintenance, mainly in the context of model driven engineering. Thus, the studied software artefacts are models, metamodels, transformations and chains. A transformation is defined with potentially several input and several output metamodels. It enables the generation of models conforming the output metamodels from models conforming the input metamodels. A chain is a sequence of transformations where the output models of a transformation are the inputs of the following transformations.

Metamodelling as support. To provide generic results, modelling or metamodelling are widely used as the fundament of the approaches proposed in this document whatever the type of software system (*i.e.* model transformation chain or traditional program). Concretely, the different artefacts (software, language, change, operator) were abstracted and reasoning is performed on these abstractions. Consequently, they can easily be adapted to other systems, languages, artefacts.

1.4 Content of this Document

This document tackles two types of software systems (model transformation chains and traditional programs) at different steps of their life cycle. Design to foresee maintenance, test, architecture modifications and co-evolution are handled. Results reported in this document rely on metamodelisation. Figure 1.1 sketches and sums up the content of this document.

Each chapter follows the same structure. First, the problem is briefly introduced. Second, a state of the art as it was when the work was realised is presented and discussed. Third, the contributions are described. Finally perspectives and conclusions are drawn. The presented results were published in international journals, conferences or workshops and readers needing more details are invited to read

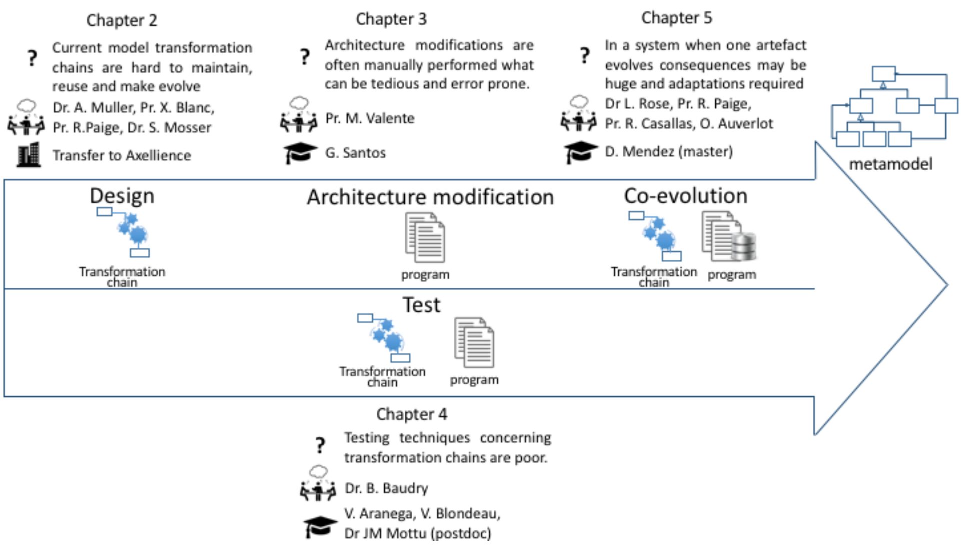

Chapter 2
?
Current model transformation chains are hard to maintain, reuse and make evolve
Dr. A. Muller, Pr. X. Blanc, Pr. R.Paige, Dr. S. Mosser
Transfer to Axellience
Chapter 3
?
Architecture modifications are often manually performed what can be tedious and error prone.
Pr. M. Valente
G. Santos
Chapter 5
?
In a system when one artefact evolves consequences may be huge and adaptations required
Dr L. Rose, Pr. R. Paige, Pr. R. Casallas, O. Auverlot
D. Mendez (master)
metamodel
Design
Transformation chain
Architecture modification
program
Co-evolution
Transformation chain
program
Test
Transformation chain
program
Chapter 4
?
Testing techniques concerning transformation chains are poor.
Dr. B. Baudry
V. Aranega, V. Blondeau, Dr JM Mottu (postdoc)

Figure 1.1: Overview of the problems tackled in the document

these papers.

Chapter 2 proposes a new way to design model transformations to enhance reusability, maintainability and scalability of transformation chains. This mostly comes out of collaborations with Dr Alexis Muller, Prof. Xavier Blanc, Prof. Richard Paige and Dr Sebastien Mosser.

Chapter 3 aims to provide support in the form of concepts, methods, techniques, and tools for different categories of architecture modifications. This chapter exposes the results conduced in the context of Gustavo Jansen Santos thesis and a collaboration with Prof. Marco Tulio Valente.

Chapter 4 focuses on data test sets. It provides generic mechanisms, in the context of model transformations to improve test data sets. It also studies problems and impact of test set selection in the context of traditional software when the test set is too large to be run entirely after a change. This chapter mainly presents results obtained (*i*) in the context of Vincent Aranega's thesis and the postdoctoral stay of Dr Jean-Marie Mottu and *(ii)* in the context of Vincent Blondeau's CIFRE thesis with Worldline.

Chapter 5 studies the co-evolution of different system artefacts. Even if the artefacts are different, the co-evolution mechanism is very similar in the two studied cases: metamodel-transformation and database schema-program. This chapter presents results conducted in the context of David Mendez' master internship and collaborations with Dr Louis Rose and Prof. Richard Paige, with Prof. Rubby Casallas and with Olivier Auverlot, CRIStAL Information System architect.

CHAPTER 2

Designing Model Transformation Chains to Ease Maintenance and Evolution

2.1 Problems

For a decade, Model Driven Engineering (MDE) has been widely applied. Large and complicated languages are used – *e.g.,* UML 2.x and profiles such as SysML[1] or MARTE[2]. Consequently, large transformations are more likely to be developed; examples have been published of transformations counting tens of thousands of lines of code. Such transformations have substantial drawbacks [Pilgrim 2008], including reduced opportunities for reuse, reduced scalability, poor separation of concerns, limited learnability, and undesirable sensitivity to changes. Other research argued that focusing on the engineering of transformations, and improving scalability, maintainability and reusability of transformations, is now essential, to improve the uptake of MDE [Wagelaar 2009] and to make transformations practical and capable of being systematically engineered [Cordy 2009].

Several classifications considered model transformations according to different criteria [Czarnecki 2003]. Here we focus on the relationship between source and target criterion that introduces two different transformation types: *in-place* or *out-place*, also called *in-out transformation*. In *in-place* transformations, the input and output metamodels are the same, the input model conforms to them and is modified in place. Refactoring transformations fall into this category. Such transformations are often very limited in terms of number of involved metamodel concepts and in terms of performed modifications. In case of *in-out* transformation, the input and output metamodels are often different, but this is not mandatory. The output model is created from scratch. To create an output model element, the metamodel concept that is instantiated has to be handled by the transformation. Traditional *in-out* transformations must manage the whole input and output metamodel. Model trans-

[1]http://www.omgsysml.org/

[2]http://www.omg.org/spec/MARTE/

formation chains are mostly composed of in-out transformation to refine details and finally generate code. Figure 2.1 sketches these two types of transformation.

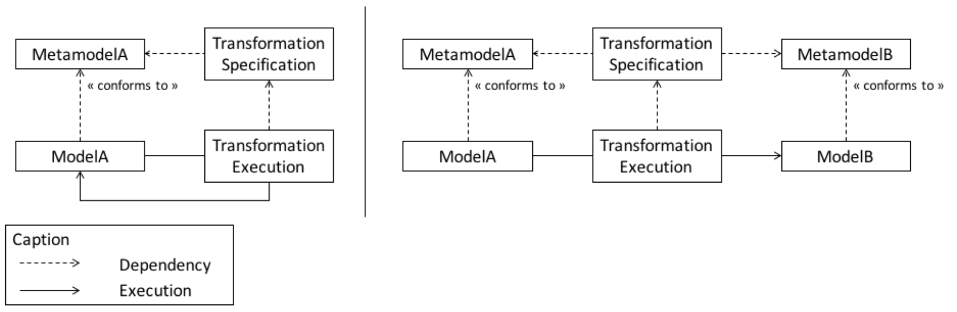


Figure 2.1: In place transformation on the left; in-out transformation on the right

Small transformations dedicated to a specific purpose are more easily maintainable or reusable than big transformations dealing with large and complicated metamodels. However, as briefly explained they have to be either in place and deal with the same input and output metamodels or in-out and manage all the metamodel concepts, that in case of big metamodels can quickly become huge. The purpose of this chapter is to tackle this contradiction. The idea is to combine advantages of both of these transformation types to get small dedicated transformations whose input and output metamodels may be different and count several hundred of concepts.

This chapter deals with model transformation chain as studied software system. It aims to enhance maintainability, reusability and modularity in this type of system. For this purpose, metamodeling techniques are used.

2.2 Previous State of the Art

This section presents the state of the art about model transformation chains as it was around 2005-2010, when the work presented here took place.

In the case of traditional systems, identification of reusable artefacts can be done from scratch or by decomposition of existing software system. In the case of model transformations the same approaches can be considered. When small transformations are built, they need to be composed into chains. First we present the existing decomposition approaches and their limits. Then we briefly introduce the composition proposals. Finally, since generic transformations also answer to the reusability requirement, we briefly explain the existing approaches and their drawbacks.

Decomposition of transformations. Hemel *et al.* describe the decomposition of a code generator (*i.e.* a model transformation chain leading to code) into small transformations [Hemel 2008]. The authors introduce two types of modularity: *vertical* and *horizontal*. Vertical modularity is used to reduce the semantic gap between input and output models. It is achieved by introducing several intermediary transformations that gradually transforms a high-level input model into an implementation. Horizontal modularity is achieved by supporting the definition of plugins which implement all aspects of a language. If vertical modularity was common already at that time as suggested by the Model Driven Architecture (MDA) process, horizontal modularity was new and it is what we wanted to tackle. Similarly, Vanhooff *et al.* highlighted the benefits of *breaking up large monolithic transformations into smaller units that are more easily definable, reusable, adaptable* [Vanhoof 2005]. In these approaches, no information is given concerning the "localised" character of the transformations. The examples of these papers only concern refactorings (*i.e. in-place* transformation with same input and output metamodels).

Oldevik provides a framework to build composite transformations from reusable transformations [Oldevik 2005]. The author assumes that a library of existing transformations is readily available. The granularity/locality degree of the transformations is not specified.

Olsen *et al.* define *a reusable transformation [as] a transformation that can be used in several contexts to produce a required asset* [Olsen 2006]. In practice, the smaller transformations are, the more they are reusable. Furthermore, the authors identify several techniques allowing the reuse of transformations such as specialisation, parametrisation and chaining. Nevertheless, no indication is provided on the characteristics of the transformations or on the way to practically and concretely reuse transformations.

Sànchez and Garcia argued that model transformation facilities were too focused on rules[3] and patterns[4] and should be tackled at a coarser-grained level [Sanchez Cuadrado 2008]. To make model transformation reusable as a whole, authors propose the *factorisation* and *composition* techniques. *Factorisation* techniques aim at extracting a common part of two existing transformations to define a new transformation. *Composition* creates a new transformation from two existing ones. Those techniques have a major drawback. They require that the intersection of the input and the output metamodels (viewed as set of concepts) is not empty.

Chaining transformations. Rivera *et al.* provide a model transformation orchestration tool to support the construction of complex model transformations

[3]As a program is composed of functions or methods, model transformations are composed of rules.

[4]The patterns define the application condition of a rule to modify the source elements (in case of in-place transformation) or to create target ones (in case of out-place transformation)

from those previously defined [Rivera 2009]. The transformations are expressed as UML activities. As such, they can be chained using different UML operators: composition, conditional composition, parallel composition and loop. Only heterogeneous transformations (*i.e.* transformations whose input and output metamodels are different) can be chained. The reuse of a transformation in different chains is thus limited by the required inclusion of the output metamodel of one transformation in the input metamodel of the next one. This approach thus has restrictions in terms of reusability and adaptability.

Wagelaar *et al.* propose the mechanism of module superimposition to compose small and reusable transformation [Wagelaar 2009]. This mechanism allows them *to overlay several transformation definitions on top of each other and then to execute them as one transformation*. This approach depends from transformation language characteristics and cannot be easily adapted to other languages.

Mens *et al.* explore the problem of structural evolution conflicts by using graph transformation and critical pair analysis [Mens 2005a]. The studied transformations are refactorings (that are *in place* transformations). The operations that a transformation can perform in such cases are precisely prescribed. Nine operations are highlighted in the paper. With such a limited number, it is possible to study in detail when the operations can be chained and, when doing so, if they are commutative.

Generic transformation. Generic programming techniques were transposed to graph transformation to increase their reusability across different metamodels [Cuadrado 2011, de Lara 2012]. For this purpose, they build generic model transformation templates, *i.e.* transformation in which the source or the target domain contains variable types. The requirements for the variable types (needed properties, associations, etc.) are specified through a concept. Concepts and concrete metamodels are bound to automatically instantiate a concrete transformation from the template. The resulting transformation can be executed as any other transformation on regular instances of the bound metamodels.

Sen *et al.* propose to define reusable transformations with generic metamodels [Sen 2012]. The actual transformations result from an adaptation of a generic transformation using an aspect based approach. A model typing relationship binds the elements of the generic metamodel and those of the specific metamodels.

These two approaches have a major drawback, the concept, or the generic metamodel must cover the whole specific metamodel to which it is bound what rarely occurs in practice.

Summary. This state of the art highlights (*i*) the requirement of reusability in transformation chains and (*ii*) the necessity to introduce a new type of transformation conjugating in-place and out place advantages.

2.3 Contributions

To enhance transformation reusability, we introduced a new type of transformation that conjugates the advantages of both in-place and in-out transformation; the *localised transformations*. This new type of transformation implies a new way to compose transformations and to build transformation chains.

2.3.1 Localised Transformations

A localised transformation [Etien 2015] applies to a tightly prescribed, typically small-in-context part of an input model; all other parts of the input model are not affected or changed by the localised transformation.

Localised transformations focus on a specific purpose, for example memory management or task scheduling in the context of massively parallel embedded system generations. Thus some elements in the input model are identical in the output model, *i.e.,* they will simply be copied over from input to output model. Manually writing such transformation logic is tedious and error prone; moreover, in the case of complicated transformations, such logic (which may be repeated in different parts of a chain of transformations) increases interdependencies and can reduce reusability and maintainability.

Thus, to increase flexibility we distinguish two parts of a localised transformation: the part that captures the essential transformation logic, and the part that copies that subset of the input model to the output model. In this way, a localised transformation can be specified with small (intermediate) metamodels only containing the concepts used and affected by the transformation. However, the models on which the transformation is executed conform to the whole metamodel, and not solely the subset on which the transformation is specified. To solve this issue, we provided an extension mechanism to extend the input and output metamodels of a localised transformation [Etien 2010]. The extension mechanism, combined with the implicit copy, provide the means to manage the transformation engineering process.

The `Extend` operator extends the notion of *in-place* transformation to transformations where input and output metamodels may be different, but the copy/identity function is implicit. Concretely, the input model is considered an instance of the union of the input and the output metamodel (1). Then, this model is copied and the transformation is executed as an *in-place* transformation on this copy (2). Finally, the model is considered an instance of the output metamodel since all the elements of the input model instantiating a concept of the input metamodel not present in the output one were consumed to produce new elements (3). Figure 2.2 presents the `Extend` operator mechanism with these three phases.

More formally, let t be a localised transformation from the source metamodel

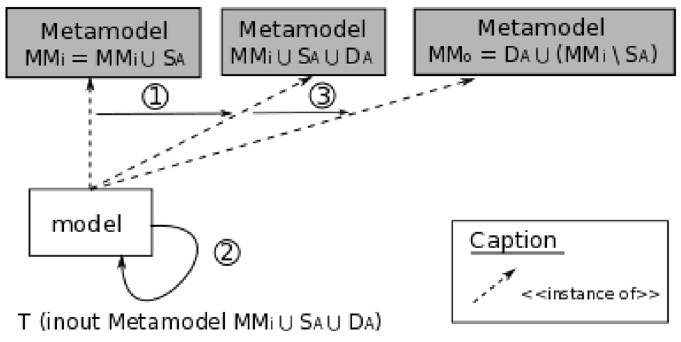


Figure 2.2: Schema of the `Extend` mechanism

S_A to the destination metamodel D_A ($t : S_A \rightarrow D_A$) and MM_i an ordinary metamodel. $\texttt{Extend}_{MM_i}(t)$ is a transformation T from the $S_A \cup MM_i$ metamodel[5] to the metamodel MM_o ($T : MM_i \cup S_A \rightarrow MM_o$), having the same behaviour that t such that:

- $MM_o = D_A \cup (MM_i \setminus S_A)$, where $MM_i \setminus S_A$ is the part of the metamodel MM_i that was not involved in the transformation, *i.e.* the part whose instances are implicitly copied
- $T(m) = t(m)$ if m is a model instance of the metamodel S_A, applying t or its extended version T is exactly the same
- $T(m) = m$ if m is a model instance of the metamodel $MM_i \setminus S_A$, m is simply copied since it does not contain elements instantiating a concept of S_A
- $T(m) = T(n) \cup (m \setminus n)$ with n the part of the m model typed by S_A. T is composed of two parts, the transformation t and the copy.

($D_A \setminus S_A \neq \emptyset$) implies that t (and thus also T) introduce new concepts not in MM_i; correspondingly, ($S_A \setminus D_A \neq \emptyset$) means that some concepts are consumed by t (and thus also T). A concept is consumed by a transformation if it exists in the input metamodel of the transformation but not in the output metamodel. Thus, the execution of the transformation aims to remove all instances of those concepts present in the input model. In theory, it is always possible to choose MM_i such

[5]We adopt the metamodel, model and conformance definitions established by Alanen *et al.* [Alanen 2008]. A metamodel is a set of classes and a set of properties owned by the classes. A model is a set of elements and a set of slots. Each element is typed as a class in a metamodel. Each slot is owned by an element and corresponds to a property in a metamodel. From these definitions, the union, the intersection and the difference are defined on both metamodels and models respectively as the union, the intersection or the difference of each set defining the metamodels or the models.

that S_A is included in MM_i; however, in practice, MM_i is not arbitrarily chosen, it depends on the chain and corresponds to the input metamodel of the chain plus (*resp.* minus) those concepts introduced (*resp.* removed) by other transformations. Indeed, if S_A is not a subpart of MM_i, this means that some concepts are useful to the execution of the transformation t but no instance will be found in any input model: another localised transformation introducing these concepts must be executed beforehand. Finally, extending t with various metamodels MM_i enables to easily reuse t.

The notion of localised transformation is an addition to the classifications established by Czarnecki and Mens [Czarnecki 2003, Mens 2005c], where the input and output metamodels are different, but with some concepts in common. Mens *et al.* distinguish heterogeneous transformations, where the input and the output metamodels are different, from endogenous transformation defined on a unique metamodel. He explicitly specifies that "*exogenous transformations are always out-place*". Czarnecki differentiates approaches mandating the production of a new model from nothing, from others modifying the input model (*e.g.* in-place transformation). These classifications do not consider sharing and copying with potentially different input and output metamodels inherent in localised transformations. The notion of localised transformation is therefore a new contribution to these taxonomies.

The notion of localised transformation is the result of a collaboration with Dr Alexis Muller and Professor Richard Paige [Etien 2015]. It has been implemented in Gaspard[6] in the context of embedded systems [Gamatié 2011]. A transfer of the transformation engine to Axellience, an Inria spin-off, occurred in 2012.

2.3.2 Composition of Localised Transformations

Once individual localised transformations have been defined, they must be composed to form a transformation chain and produce the expected result. Defining this composition is not trivial: if the input metamodel for the chain is known, the order in which localised transformations are executed has to be calculated precisely, since some orderings do not lead to models conforming the output metamodel (*e.g.,* by leaving an intermediate model in an inconsistent state that cannot be reconciled by any successive subchain of localised transformations).

Traditionally, input and output metamodels are either completely separated or form only one. We formally defined rules to identify valid compositions of transformations [Etien 2010]. These rules can be applied to localised transformations, since they consider transformations whose input and output metamodels overlap.

[6]Gaspard is a hardware/software co-design environment dedicated to high performance embedded systems based on massively regular parallelism. It has been developed at Inria Lille Nord Europe by the Dart team to which I belonged (https://gforge.inria.fr/frs/?group id=768).

They rely on a structural analysis of the small metamodels involved in each localised transformation. We briefly summarise this here.

Consider two localised transformations, $t_A : S_A \rightarrow D_A$, and $t_B : S_B \rightarrow D_B$.

Definition: Chaining of localised transformations. t_A and t_B , can be chained if there exists a metamodel MM_A on which the first transformation can be extended using the `Extend` operator and if the concepts used by the second transformation are included in the output metamodel of the first extended transformation. More formally, $\exists\, MM_A$ such as $S_B \subseteq D_A \cup (MM_A \setminus S_A)$. This inclusion implies that the concepts used by the second transformation (t_B) are not consumed by the first one (t_A) *i.e.* $S_B \cap (S_A \setminus D_A) = \varnothing$.

From this definition, it is possible to extract the following property:

Property: Chaining of extended transformations. If t_A and t_B can be chained then their extended version T_A and T_B can also be chained corresponding to the classical $T_A \circ T_B$, with $T_A = Extend_{MM_A}(t_A)$ and $T_B = Extend_{D_A \cup (MM_A \setminus S_A)}(t_B)$

The input metamodels S_A and S_B are subsets of MM_i with $MM_i = MM_A \cup S_A$ (plus eventually other concepts created by other localised transformations). Three cases may occur:

1. *t_A and t_B can only be combined in one order (t_A then t_B for example).* This means that t_A can be chained with t_B or t_B can be chained with t_A.
2. *t_A and t_B can be combined in both orders.* The order of the two localised transformations t_A and t_B can be swapped if t_A can be combined with t_B and vice-versa *i.e.* the chaining definition is applied in both orders. But we cannot guarantee that in both orders, the resulting models are equivalent. If the input metamodels of the two transformations t_A and t_B have no common elements and if the concepts required by t_A (respectively t_B) are not produced by t_B (respectively t_A), they can be combined and the resulting model does not depend on their execution order. If $(S_A \cap S_B) = \varnothing$ and $(D_A \cap S_B) = \varnothing$ and $(D_B \cap S_A) = \varnothing$ then, for all models m, chaining extended versions of the transformations t_A and t_B leads to the same result than chaining them in the opposite order.
3. *t_A and t_B cannot be combined at all.* The combination of t_A and t_B transformations is impossible when each transformation consumes concepts useful for the execution of the other *i.e.* if $S_B \cap (S_A \setminus D_A) \neq \varnothing \ and \ S_A \cap (S_B \setminus D_B) \neq \varnothing$.

This study on the chaining of localised model transformation results from a collaboration with Dr Alexis Muller and Professor Xavier Blanc [Etien 2010], [Etien 2012].

2.3.3 Building Model Transformation Chains

To introduce flexibility and reusability, the Gaspard environment [Gamatié 2011] has been re-engineered to rely on localised transformations. Each transformation has a single intention such as memory management or scheduling and corresponds to 150 lines of code in average. 19 transformations including 4 *model to text* (M2T) transformations, and 15 *model to model* (M2M) transformations were defined. The number of chains that can be constructed from them is huge, (bigger than $6{,}5 \times 10^{12}$). But only a few chains make sense. It becomes crucial to help the designer to built such chains. Thus, the definition of transformation libraries raises new issues such as *(i)* the representation of the transformations highlighting their purpose and the relationships between them; *(ii)* their appropriate selection according to the characteristics of the expected targeted system and *(iii)* their composition in a valid order.

To tackle the aforementioned issues, we proposed, in [Aranega 2012], a feature-oriented approach and the associated tool set to automatically generate accurate model transformation chains as depicted in Figure 2.3. Since a localised transformation has a specific intention, it is possible to define a feature diagram where each leaf feature corresponds to one of the intentions introduced by a localised transformation. Intermediary features enable the classification.

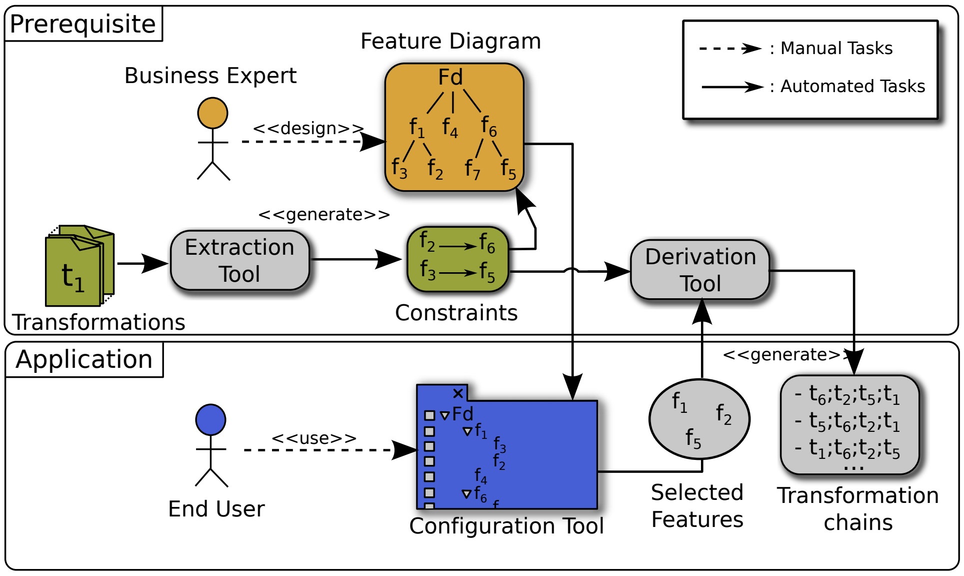


Figure 2.3: Approach Process Overview (copied from [Aranega 2012])

This approach relies on three pillars: *(i)* the classification of the available transformations as a *Feature Diagram* (FD) produced by the *business expert*[7], *(ii)* the

[7]The business expert knows the domain and the transformations

reification of requirement relationships between transformation (directly generated from the *Transformations* set by the *Extraction Tool*) and *(iii)* the automated generation of transformation chains for a given product (using our *Derivation Tool*) from features selected by the *end user*.

The FD is designed once for all by the *business expert* as a prerequisite. It is nevertheless possible to modify it when new transformations and thus new features become available. The requirement relationships are expressed between the features and automatically computed from the transformation codes by the *Extraction Tool* we provide. The extracted relations enable to derive dependent features (and then the associated transformation) from the ones selected by the designer using a *Configuration Tool* (*e.g.,* FeatureIDE[2]). The requirement relationships are also used by our *Derivation Tool* to order the selected features design valid chains.

This work results from a collaboration with Dr Vincent Aranega and Dr Sébastien Mosser [Aranega 2012].

2.3.4 Localised Transformations Characteristics

Time saving. Building the first chain (*i.e.* designing the first localised transformations because no one is already available on the shelves, and then chaining them) takes approximately the same amount of effort as building a non-localised transformation chain. Indeed, in traditional approaches, the intermediate metamodels have to be defined, the transformations between them written and validated and the resulting system has to be tested. The involved metamodels are often large, leading to transformations that can be difficult to specify and to test. In our approach, the number of metamodels and transformations is greater but the complexity to specify, test and validate each of them is reduced.

However, the time for the development of the next chains is reduced depending on the number of reused transformations. In Gaspard, this time has been reduced to around 25% for the different chains we built then. Indeed, only the transformations dedicated to the new target and the code generation have to been developed. Such an improvement does not generally exist for traditional approaches since transformations are not easily reusable.

Reusability. Using localised transformations is valuable in the context of developing a family of related transformations. Transformations constitute a family when they exhibit similarities and variabilities. Such a context occurs, for instance, when various technologies are targeted from the same core source language, like in Gaspard [Gamatié 2011] where OpenMP, OpenCL, pThread and SystemC code

[2]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

are generated from the MARTE metamodel, or in information system design if the J2EE technology and .NET technology are each being targeted.

The reusability of the chains is very high within a domain when using localised transformations. However, reusability is reduced between chains of different domains. The introduction of genericity in localised transformations is likely to enhance reusability.

Test. With our approach, testing and validating model transformations require less effort, because generally each building block of the model transformation is smaller. Indeed, each localised transformation has a unique and very specific intention. By comparison with large transformations, it is thus easier to check that the transformation does what is expected or not. Furthermore, thanks to the localised characteristic of our transformations, it should be possible to perform a test fully covering the metamodels. Indeed, one of the crucial issue in test activity is to qualify the input data *i.e.* the ability of the data set to highlight errors in a program or a transformation.

Modularity and Understandability. By analogy to the work presented by van Amstel [van Amstel 2008], we consider that the modularity of a transformation chain positively depends on the number of transformations and negatively depends on the unbalance (*i.e* module size compared to the average of module size) and the number of rules or queries by transformation. In essence, using localised transformations increases the number of transformations and decreases the unbalance and the number of rules by transformation.

According to van Amstel, *"A large number of modules is no guarantee for an understandable model transformation. The modules should be balanced in terms of size and functionality."* [van Amstel 2008]. Similarly, we can affirm that a large number of transformations is no guarantee for an understandable transformation chain. However, localised transformations are in essence very small, focus on a single intention and work with very small metamodels whereas the traditional input metamodel is large with hundreds of concepts like UML or one of its profile. Thus, each localised transformation is more easily understood than a traditional transformation and by transitivity also the chain it self. In fact, the complexity is transferred to the composition of the chain in order to ensure that the localised transformations can be chained and fulfill the specifications.

2.4 Perspectives

The work presented in this section has been performed when model transformation tools and languages started to become mature. Examples in the articles were only toy examples. Gaspard was one of the first environment using transformations for real. This concrete case study enables us to meet new issues relative to transformation and chain maintenance. Solving them leads to the introduction of localised transformations that brought some new challenges.

Construction, Decomposition. The concept of localised transformation has been introduced, to enhance reusability and ease maintenance. We provide mechanisms to compose and to build chains from such transformations available on the shelves. However, we gave no indication on the way to specify such transformations. They can be either defined from scratch or by decomposing existing "traditional" transformations. Similarly to component based approaches dedicated to software programs, finding the most appropriate size of a localised transformation component, migrating from a classical transformation chain to one using localised transformations are issues that remain unsolved and would be relevant to tackle.

Chaining localised transformations. We identified some chaining rules based on the inclusion of the metamodels of the localised transformation in the extended metamodels [Etien 2010]. We also highlighted that if, according to the metamodel inclusion, some transformations can be switched the produced models can be different. Consequently, several transformation chains are syntactically correct but potentially not semantically. New constraints concerning functionality and business have to be checked. A new abstraction level providing more information relative to the intention and the output of the transformations has to be defined. It should be independent of the used transformation language, and consider the transformation as a black box. This new level will allow a more fine-grained analysis relative to the typing constraints.

Towards Genericity. We claim that our approach is context independent. However, a localised transformation is integrated in a transformation chain only if its input metamodel is included in the extended metamodel of the previous transformation. Such a chaining condition involves a dependency of the localised transformations with the initial input metamodel.

In fact, the localised transformation concept is a first indispensable step towards generic transformation. The localised transformations coupled to the `Extend` operator enable a definition on small input metamodels, and an execution on models conform to much larger ones. To introduce genericity in transformations, mechanisms like templates have to be associated to those presented in this chapter. Exploring this research track would enable us to define localised transformation that could be more largely used than only in the context of one single input metamodel.

CHAPTER 3

Supporting Software Architecture Modifications

3.1 Problems

Software systems must constantly evolve for example to fix bugs, adapt a system to accommodate API updates, improve systems structure or answer new user requirements.

Tools exist to repair bugs, refactor a code or accommodate API updates. Often implied modifications are confined, mostly inside a single method or a class. These modifications occur daily. However, during their lifecycle, systems meet other types of modifications for example splitting a package in two, moving classes, introducing new abstractions, or reorganizing classes. No tool support such larger changes possibly implying several methods, classes or even packages and that are considered architecture modifications.

Avgeriou *et al.* distinguish three types of approaches for systematically handling architecture changes, listed in an order of increasing severity: *refactoring*, *renovating*, and *rearchitecting* [Avgeriou 2013].

- *Architecture refactoring* is larger than what is supported by the IDE (and corresponds to code refactoring), since it can correspond to dependency cycles or overly generic design. Such changes require medium effort. They are regularly performed during system lifecycles and focussed on some components to modify them.
- *Renovating* is complementary to refactoring because it also deals with only parts of the system. It consists in the creation of new components from scratch. It occurs less frequently than refactoring.
- Finally, when an architecture is subject to significant changes, refactoring or renovating won't always suffice. This might be the case when a technology platform is replaced by a newer one, when there is a significant change in business scope, or when the architecture is in such bad shape that errors keep emerging. In such cases, *rearchitecting* is necessary. It corresponds to substantial modifications implying the whole system. Components are reused, modified or built.

Such activities suffer from the absence of concepts, methods, techniques and tools. In this chapter, we tackle rearchitecting and architecture refactoring. Admittedly, it is the two extremes, but both consider existing components that are reused or modified. We provide a solution to enable architects to easily check constraints on different versions of the system, and another to restructure systems at a finer grain by system specific transformations.

3.2 Previous State of the Art

Architectural Restructuring and Constraint Validation. That *et al.* use a model-based approach to document architectural decisions as architectural patterns [That 2012]. An architectural pattern defines architectural entities, properties of these entities, and rules that these properties must conform to. The approach provides analysis by checking the conformance between an existing architecture definition and a set of user-defined architectural patterns.

Baroni *et al.* also use a model-based approach and extend it to provide semantic information [Baroni 2014]. With assistance of a wiki environment, additional information is automatically synchronised and integrated with the working model. The analysis consists in checking which architectural entities are specified in the wiki. One critical point of this approach is that the information might be scattered in different documents, which can be difficult to maintain.

Definition of Composite Transformations. Several authors propose to introduce design patterns in existing software systems by application of transformations [France 2003, Kim 2013] . For this purpose, they specify (i) the problem corresponding to the design pattern application condition, (ii) the solution corresponding to the result of the pattern application and (iii) the transformation corresponding to the sequence of “operation templates” that must be followed in order for the source model to become the target model.

Other work also defined transformation patterns by application condition and operators [Lano 2013], based on temporal logic [Mikkonen 1998], and based on graph transformation [Mens 2007].

Such work defined transformations for a very generic purpose, *e.g.,* to daily modify models. These transformations may often be automatically applied. They cannot be applied to automate repetitive tasks during an architecture modifications.

Change Operators. Javed *et al.* categorise change operators on source code in three levels, described as follows [Javed 2012]. **Level one** operators are atomic and describe generic elementary tasks. For example, these operators are routinely proposed in IDE like ECLIPSE as development helpers (*e.g., Extract Method*), and calculated from source code in the CHANGEDISTILLER tool [Fluri 2007]. These operators are generic in the sense that they are independent of the system, the application domain, and sometimes even of the programming language. **Level two** operators are aggregations of level one operators and describe more abstract composite tasks. For example, the *Extract Method* is a composition of several atomic changes (*e.g., Create Method*, *Add Statement*, etc.). These operators depend on the programming language they are based on. However, they are still generic because they can be applied to systems from different domains. Finally, **level three** operators are aggregations of level one and level two operators, and they are domain specific. This classification relies on two major characteristics, the size of the change operators (atomic versus complex) and the application domain (generic versus domain specific).

Code Refactoring as Repetitive Source Code Transformations. Developers and researchers alike have long perceived the existence of repetitive source code transformations. This led them to propose some automation of these transformations, in order to reduce mistakes and ease the work of developers. As a consequence, integrated development environments (such as ECLIPSE) include refactoring transformations as a way to automate composite transformations that define behavior-preserving tasks. They are inspired by the refactoring catalog proposed by Fooler [Fowler 1999].

However, recent work proved that code refactoring tools are underused. Two different studies based on the code refactoring tools proposed by ECLIPSE platform were conducted [Murphy-Hill 2009] and [Negara 2013]. Both studies lead to the conclusion that, when a code refactoring transformation is available for automated application, the developers prefer to manually perform the transformation. Similar results based on both a survey and a controlled study with professional developers were observed [Vakilian 2013]. Developers do not understand what most of operators proposed by code refactoring tools do, or they do not perceive how the source code will actually change after their application. Therefore, developers prefer to perform a sequence of small well-known code refactoring transformations that will produce the same outcome as a composite, sometimes complex, built-in code refactoring transformation. There is thus a real need for the developers to understand the modifications they are automatically applying.

Architecture Refactoring Performed through Source Code Transformations. Fluri *et al.* propose a clustering approach to identify types of code changes that occur together and repeatedly inside methods [Fluri 2008]. The authors categorise these changes by the semantics of their activities.

In a different scope, AST differencing and association rule mining are used to recommend candidate files to change based on similar changes in the past [Ying 2004]. The approach generates recommendations that can reveal subtle dependencies across files that are not clear in the code.

Jiang *et al.* considered the system specific property of changes [Jiang 2015]. Their contribution consists in considering that a transformation may involve separate changes during time. The authors identified patterns in real-world systems and categorise them by the semantics of the development changes. They observed that some types of task take days and several developers to be completed.

Summary. This state of the art highlights (*i*) the absence of tool to validate constraints in the context of rearchitecting, (*ii*) the fact that architecture refactoring can be achieve by a sequence of operators on the code and (*iii*) the need for developers to understand the transformation they are automatically applying.

3.3 Contributions

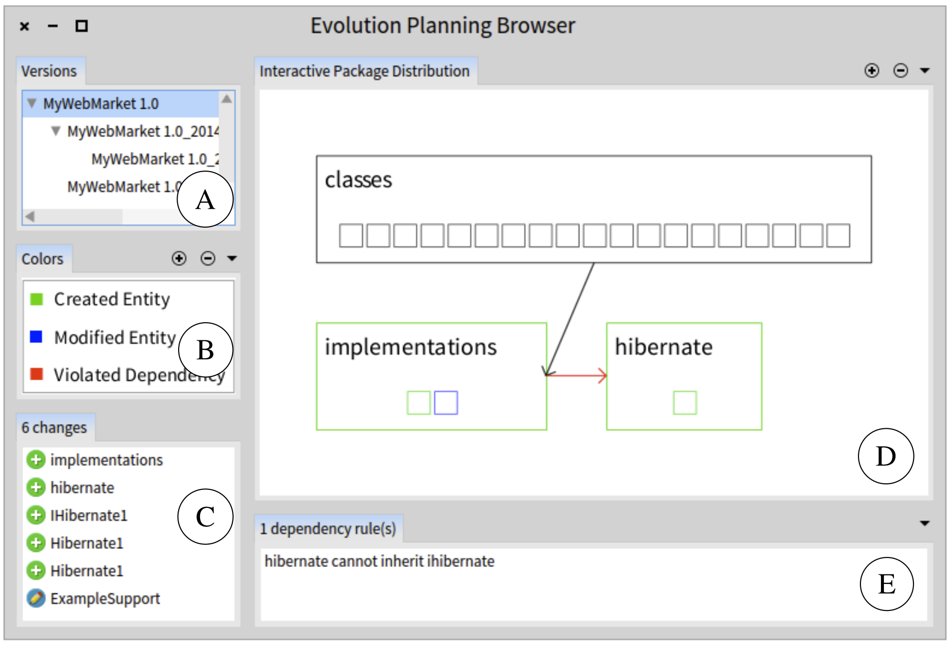
3.3.1 Architectural Modifications and Constraint Validations

To help architects to describe systems and validate architectural constraints, we developed ORIONPLANNING [Santos 2015a]. ORIONPLANNING relies on (i) FAMIX [Ducasse 2011], a family of meta-models to represent source code entities and relationships of multiple languages in a uniform way; (ii) MOOSE[1] for the metric definition such as size, cohesion, coupling, and complexity metrics; and (iii) ORION, a reengineering tool to simulate changes in multiple versions of the same source code model.

ORIONPLANNING enables users to define an architecture from scratch or iteratively modifying a current architecture extracted from source code. Several alternative versions can be explored and analysed. Our tool provides graphical representations of the system at various granularity levels (package, class or method) to assist the modification of an architecture. Architecture modifications can be performed on these representations and color code enables the identification of changed entities per version and per type of change. ORIONPLANNING also proposes an analysis environment to check whether a given architecture (*i.e.,* the current one or one of its various versions) is consistent with user defined restrictions. These restrictions are written as rules based on the existing metrics *e.g.,* restricting the number of classes in a package to less than 20. Our prototype also supports the definition of dependency constraints. The definition uses the syntax of DCL [Terra 2012], a domain specific language for conformance checking.

Figure 3.1 depicts the main user interface of ORIONPLANNING. Panel A shows

[1]http://moosetechnology.org/



hibernate cannot inherit ihibernate

Figure 3.1: ORIONPLANNING overview.

the system under analysis and its versions, followed by a panel for color captions (Panel B), and the list of model changes in the selected version (Panel C). On the right side of the window, ORIONPLANNING provides a visualisation of model entities and dependencies (Panel D) and a list of dependency constraints which will be evaluated when the model changes (Panel E).

3.3.2 Transformation Pattern Definition

Architecture refactoring often consists in repetitive code transformations over several components. These transformations are not mandatorily behavior preserving. Moreover, even if they are performed several times in the system, they are specific to it. Concretely, they consist in a sequence of operators applied on different entities.

Motivating Example. Listings 1 and 2 illustrate an example of repetitive source code transformation extracted from PACKAGEMANAGER, a package management system for PHARO[2]. They present code edition examples in two distinct classes, named `GreasePharo30CoreSpec` and `SeasideCanvasPharo20Spec`. For comprehension purposes, we illustrate all the code examples in a Java-inspired

[2]http://pharo.org/

syntax. We also represent changed parts of source code in terms of added (+) and removed (−) lines.

Concerning the transformations involved, the developers removed a method named `platform()`. This method defines: (*i*) on which IDE configuration the current package depends, and (*ii*) the name of the package in its repository. This definition is made by invocations to the methods `addPlatformRequirement` and `addProvision`, respectively. Instead, the developers updated this definition so that each package "*only provide data and do not call methods*"[3]. In this way, the developers created two methods, named `platformRequirements()` and `provisions()`. Both of them return an array of strings, containing the same arguments as in the `platform` method, that is removed. This new definition is more similar to a package manifest.

Listing 1: Modified code in `GreasePharo30CoreSpec`

```
− public void platform() {
−     package.addPlatformRequirement("pharo");
−     package.addProvision("Grease−Core−Platform");
− }

+ public String[] platformRequirements() {
+     return { "pharo" };
+ }

+ public String[] provisions() {
+     return { "Grease−Core−Platform" };
+ }
```

These transformations impact three methods of one class. Although these transformations seem simple, they were performed on 19 distinct classes. They are part of an architecture refactoring. Specifically, the transformations apply to all classes that extend the class `PackageSpec` and define a method named `platform()`. Other few classes, which are responsible for deserializing the package definitions, were transformed as well since the method `platform()` was removed. However, their updates related to this transformation are not repetitive and therefore they are not considered in this discussion.

Definition: An *application condition* selects, from all the entities in a system (e.g., classes, methods, etc.), which ones must be transformed.

Listing 2 shows the result of the same transformations, this time performed in the class `SeasideCanvasPharo20Spec`.

[3]We found this commit message in PACKAGEMANAGER's version control repository.

Listing 2: Modified code in `SeasideCanvasPharo20Spec`

```
– public void platform() {
–     package.addPlatformRequirement("pharo2.x");
–     package.addProvision("Seaside–Canvas–Platform");
– }

+ public String[] platformRequirements() {
+     return { "pharo2.x" };
+ }

+ public String[] provisions() {
+     return { "Seaside–Canvas–Platform" };
+ }
```

As any algorithm, each transformation requires some specific information to be executed. For example, to perform an *Add Method* transformation, one must provide the signature of the method, and the class in which this method will be added. We call this information, the *parameters* of the transformation.

Definition: A *parameter* is an input, e.g, a variable or a value, that is necessary for a transformation to be executed.

Definition: The *context* of a set of transformations is the collection of parameters that are needed to execute its containing transformations.

Table 3.1 roughly summarises the contexts in these two examples. More specifically, some parameters are (i) similar in both transformations, e.g., the signatures of the (removed and added) methods are the same in Table 3.1. However, some parameters are (ii) non-identical, e.g., the return statements in Table 3.1 vary from one class to the other one. Therefore, just performing the transformations as they were defined in the first example would not produce the desired output in the second one.

Table 3.1: Context required to perform the transformations in the classes `GreasePharo30CoreSpec` and `SeasideCanvasPharo20Spec`, as presented in Listings 1 and 2.

Transformation	`GreasePharo30CoreSpec` (as seen in Listing 1)	`SeasideCanvasPharo20Spec` (as seen in Listing 2)
Remove Method	`platform()`	`platform()`
Add Method	`platformRequirements()`	`platformRequirements()`
Add Return Stat.	`{ "pharo" }`	`{ "pharo2.x" }`
Add Method	`provisions()`	`provisions()`
Add Return Stat.	`{"Grease-Core-Platform"}`	`{"Seaside-Canvas-Platform"}`

Transformation Pattern Definition. Based on these definitions, we define the notion of *Transformation Pattern*. The term *pattern* comes from repetition of code transformations.[4]

Definition: A *transformation operator* is a code transformation that can be atomic or aggregated, i.e., it considers transformations of levels one and two.

Definition: A *transformation pattern* is composed of (i) an application condition and (ii) a sequence of transformation operators.

The operators are ordered because they are dependent from each other [Mens 2007].

Listings 1 and 2 showed the result of the code transformations in a text based format. We represent the same example in terms of code transformations in Pattern 3.1. It is worth noting that the representation of Pattern 3.1 is purely to understand the transformations that took place. Transformation patterns are not represented like this in our approach.

PATTERN 3.1: PACKAGEMANAGER's transformation pattern.

Description: Correcting package platform definition
Applied to: 19 classes.
Condition: ∃ class `C` that extends `PackageSpec` and ∃ method `M` in `C` named "platform"

1. *Add Method* `M′` named "platformRequirements" in `C`
2. *Add Return Statement* in `M′` with an array containing:
   the argument of the invocation to "addPlatformRequirement" in `M`
3. *Add Method* `M″` named "provisions" in `C`
4. *Add Return Statement* in `M″` with an array containing:
   the argument of the invocation to "addProvision" in `M`
5. *Remove Method* `M`

In this pattern, each step (lines 1 to 5) consists in a transformation operator. These transformations are exactly the ones presented in Table 3.1. The application condition specifies that this transformation pattern shall be applied to all of the classes extending `PackageSpec` which implement a method named `platform()`. Moreover, each transformation operator requires some parameters to be assigned, e.g., class `C`. We provided examples of the context of the transformation pattern as shown in Table 3.1.

3.3.3 Relevance of Transformation Patterns

We propose research questions to discuss the importance of automated support in the application of transformation patterns. We restrict our study to system specific code transformations. We presented one example of transformation pattern in the

[4]From Merriam-Webster dictionary, *the regular and repeated way in which something happens or is done* [dic].

previous section. Although there are evidences in the literature of the existence of such transformations [Nguyen 2010, Ray 2012, Nguyen 2013], there is a lack of approaches that provide support for composite, system specific transformations. Considering this specific context, we propose a main research question:

RQ1 *Can we identify instances of (system specific) transformation patterns in other systems?*

Assessing Transformation Patterns. We propose RQ1 to demonstrate the generality of the problem. To complement this research question, we also evaluate potential properties of the transformation patterns that motivate some automated support in their application. Note that we will not further formalise our research questions (formal hypothesis) or formally test them. All that is required in this study is proof of existence in various systems. We describe the complementary research questions as follows.

CRQ1 *Are transformation patterns applied to all of the transformation opportunities?* For each application condition, we investigate whether the corresponding transformation pattern was applied to all of the code entities it was supposed to.

CRQ2 *Are transformation patterns applied accurately in each code location?* Given that a transformation pattern is a sequence of operators, we investigate whether all of the operators were performed in each occurrence of the pattern.

CRQ3 *Are transformation patterns applied over several revisions of the system?* We investigate whether the patterns were applied at once or over several revisions.

These research questions were evaluated on four Java programs: ECLIPSE, JHOTDRAW, MYWEBMARKET and VERVEINEJ; and five Pharo systems that underwent a restructuring effort in our research group: PETITSQL, PETITDELPHI, PACKAGEMANAGER, TELESCOPE and GENETICALGORITHM [Santos 2015c]. Table 3.2 gathers the values for the metrics relative to the different research questions. TELESCOPE and GENETICALGORITHM for which no pattern was identified do not appear in the table.

Target Systems (RQ1) We identified transformation patterns in seven out of nine systems. These systems use two different programming languages (Java and Pharo), and our study analyzed only one specific version of each system, related to their rearchitecting. We identified more than one pattern in two systems.

Are transformation patterns applied to all of the transformation opportunities? (CRQ1) Three out of eleven transformation patterns were not applied to all the

Table 3.2: Descriptive metrics of transformation patterns in our dataset

Transformation patterns	Application conditions	Pattern occurrences	Number of operators	Number of parameters
Eclipse I	34	26	4	3
Eclipse II	86	72	1	1
JHotDraw	9	9	5	2
MyWebMarket	7	7	5	3
VerveineJ	3	3	2	2
PetitDelphi	21	21	2	3
PetitSQL	6	6	3	6
PackageManager I	66	66	2	7
PackageManager II	19	19	3	5
PackageManager III	64	64	2	4
PackageManager IV	7	7	3	5

opportunities matching the application condition. When the patterns covered all the opportunities, this fact was due to their low frequency, or because the pattern consisted of a systematic and corrective task.

Are transformation patterns applied accurately in each code location? (CRQ2) In one out of eleven transformation patterns, not all of their transformation operators were performed in some occurrences. This fact does not seem to be correlated with the number of transformation operators, neither with the number of occurrences of the pattern.

Are transformation patterns applied over several revisions of the system? (CRQ3) Two out of eleven transformation patterns were applied in several revisions. This fact might be related to the perfective maintenance nature of their transformations, i.e., not applying the transformation pattern in all the occurrences did not seem to have impact on these systems.

3.3.4 Automating Transformation Pattern Application

Transformation patterns may be complex and possibly applied in a lot of different occurrences. The previous evaluation highlighted that some occurrences were incomplete, completely missing, or identified through several later revisions. To ease the application of transformation patterns, we provide an automated support.

MacroRecorder has been developed to record, configure, and replay transformation patterns [Santos 2015b]. Using our approach in practice, the developer manually performs the changes once. The tool collects and stores these changes. The developer then specifies a different code location in which MacroRecorder must replay the recorded changes. The tool generalises the recorded changes into a customised transformation that would be instantiated in the specified location and automatically configure it. In some cases, this generalisation has to be manually edited or performed. Finally, the tool searches for fragments of source code that *match* the customised transformation specified in the previous stage. If successful, the tool instantiates the transformation into these code entities and performs the transformation automatically.

The current implementation of the tool is developed in PHARO. It relies on the following requirements:

- a code change recorder. The recorder is an extension of an IDE (e.g., ECLIPSE, EPICEA for PHARO) which is responsible for monitoring activity edition and storing code changes through operators;
- an IDE supporting source code entities inspection and automatic manipulation of their underlying code (for parameter automatic configuration);
- a code transformation tool (e.g., ECLIPSE's refactoring tools, REFACTORING in PHARO). The transformation tool will be extended to provide replication of each recorded code change event.

MACRORECORDER relies on an abstract representation of the code and on a change metamodel. Figure 3.2 presents an overview of the proposed approach (highlighted in grey). The transformation operator establishes the connection between recorded code change events in EPICEA and code edition algorithms in the transformation tool (through *ParamResolver* that resolves parameters). A transformation pattern is a special type of operator that contains (and eventually executes) a collection of transformation operators.

This work is realised in the context of Gustavo Santos PhD thesis that I co-supervise with Dr Nicolas Anquetil and within a collaboration with Professor Marco Tulio Valente from Universidade Federal de Minas Gerais, Brazil.

3.4 Perspectives

Back to Code. ORIONPLANNING enables the architect to modify software architectures on a large scale based on graphical representations and constraint verifications. Several alternatives can be explored and analyzed. Currently, modifications occurring on the abstract representation of the software have to be manually applied on the code. However we believe that such manual activity is tedious and error prone. The goal is to generate code snippets, following the assumption that the architecture might not be fully described. The snippets would have enough information for developers to further complete them. An important improvement would be to include Abstract Syntax Tree (AST) modeling to ORIONPLANNING, in order for it to handle more fine-grained operators (*e.g.,* Extract Method).

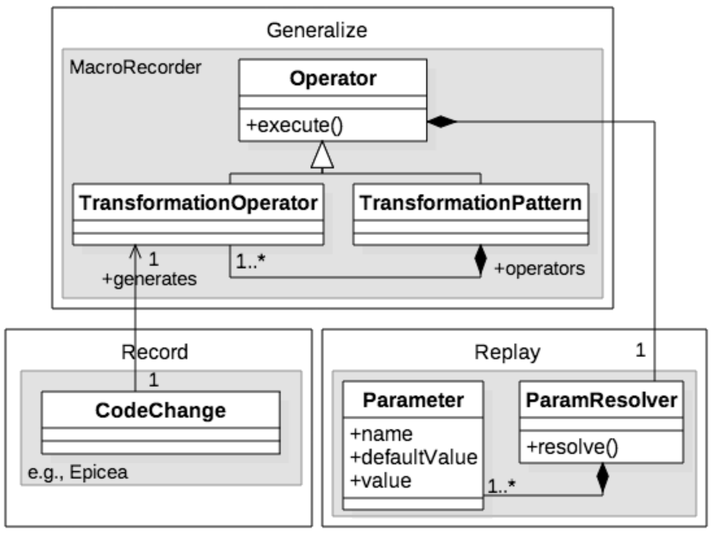


Figure 3.2: Overview of MACRORECORDER approach

This representation opens new research perspectives such as the opportunity to tackle language transformations, for example to switch from Java to Pharo. Paradigm changes such as from Cobol to Java would be managed later when transformation inside the same family of languages is mastered.

On the other hand, the introduction of an AST representation of the code will largely increase already very big models. Scalability issues have to be foreseen. A solution could be to have access to this finer representation on demand. It also raises new issues since rearchitecting is often performed on abstract system representations without taking into account fine grained information contained in the code.

Automating Transformation Pattern Application. With only one or two examples of the pattern application its is hard or even impossible to deduce the application condition. Consequently, the developer has to select new location, before the parameters are automatically matched and the transformation pattern applied again. Selecting one by one these new locations can be tedious for example when they are 72 as in one of the studied example. An alternative is to manually specify the application condition to enable a wide application on the whole system. Deducing or tuning the application condition would help in the diffusion of such tool.

Deducing Patterns from Activities. In its current version, MACRORECORDER has to be explicitly launched to record the first application of the pattern. We can imagine that soon, the tool will work in background and will analyse the events to detect patterns and then propose them to the developers when it is the third time they are performing the same sequence of operators at different places of the system.

Such a functionality implies to determine what is the size of a pattern. A pattern with a single operator can be played with various parameter values very often. It may not have much sense and in fact no even correspond to the notion of transformation pattern. A too long pattern will never be applied several times. Moreover, some times, the order of some operators can be switched without any consequence. But the tool will not discover a transformation pattern if it is looking for the exact sequence of operators. Finally, sometime when modifying the system architecture, the developers may be interrupted and do something else in the system without link to the pattern and go back to it. Once again in these conditions the discovery of the patterns is even more complex.

CHAPTER 4

Testing Supported by Metamodelling

4.1 Problems

In the context of traditional systems, errors observed during the execution may come from the compiler or the source program. In an MDE context, such a distinction can be established, between errors in the transformation definition and errors in the source model. Errors in transformations may have huge consequences. Indeed, transformations are used many times to justify the efforts relative to their development. So if they are erroneous, they can spread faults to models several times. Consequently, as any program, but also for these reasons, model transformations need to be tested.

Obviously, model transformation may be considered program and consequently tested. However, existing approaches do not take into account the specific features of model transformations, *i.e.* (i) the three fundamental operations composing them (navigation, filtering and creation or modification of new model element) and (ii) models as input data. Traditional testing approaches have thus to be adapted to model transformations. Such adaptations may be performed for each specific transformation language / approach or in the opposite may take into account their heterogeneity. In the work presented in this chapter we chose the second alternative that relies on the common features of the transformations.

Several problems need to be solved when tackling model transformation testing. First, we need to detect the presence of errors by observing wrong execution of the model transformation. Some corresponding challenges are (*i*) efficient test data production and (*ii*) observation of error in the system. This first point consists in the production of new test until reaching a satisfaction threshold. This latter point corresponds for example to the comparison of the effective output model with an oracle, the expected output model. Only the first challenge is discussed in this chapter. We considered the error as observed in the output model. Then we have to locate the error in the transformation and to fix it.

Producing efficient data test may then lead to the management of large test sets. When test data sets are large, running all the tests may take hours. Consequently, they are not launched as often as they should and are mostly run at night. Worldline, a major IT company wishes to improve its development and testing process by giving to developers rapid feedback after a change. An interesting solution is to reduce the number of tests to run by identifying only those exercising the piece of code changed. Before deploying a test case selection solution, Worldline, called us to investigate the situation in its projects and to evaluate different approaches on three industrial, closed source, cases to understand the strengths and weaknesses of each solution. Projects provided by Worldline are mixed (traditional and using model driven engineering techniques). We tackled this issue by focussing on traditional programs.

4.2 Previous State of the Art

Testing model transformation is quite similar to testing software [Xanthakis 2000]. Synthetically, it corresponds to execute a program with an input data test set and to check if the obtained results are the expected ones. If it is not the case, this test detected an error that needs to be corrected. After the fix, the process is restarted until the test succeeds. Completely testing a software system is very difficult since it means to foresee all the possible alternatives. Consequently, a software system is only tested up to a satisfactory level. This level is evaluated through criteria like test coverage.

Test Model Automatic Generation. Küster *et al.* define a template language relying on input and output metamodels of the tested transformation as well as its rules structure to automatically generate test models [Küster 2006]. This approach relies on a white box representation of the transformation (*i.e.* its implementation is known). It is dependent on the used transformation language. To be used on transformations written in an other language, adaption or new definition are required.

Sen *et al.* build test models that conform to a given metamodel by combining constraints written in Prolog [Sen 2007]. However, Prolog does not constraint enough model creation. An alternative with Alloy [Jackson 2002] is provided by the authors [Sen 2008].

Another technique to automatically generate tests is proposed by Ehrig *et al.*. A graph grammar is derived from a metamodel and traditional techniques of test generation from grammars are then used [Ehrig 2009].

Guerra *et al.* tackle the test model generation challenge by deriving, from the transformation specification, a set of test models ensuring a certain level of coverage of the properties in the specification [Guerra 2012]. These input models are computed using constraint solving techniques.

Because these approaches rely on static analysis of the transformation, if there is some dead code, tests are generated for these parts, even if they are never executed. It can be a problem, since metrics like test coverage compute a data for the whole program dead code included. Because the transformation will not be fully covered by test it seems better to focus on parts used for real. Moreover, an important question is raised concerning the required size of the test model set and its quality.

Test Model Qualification. Test model qualification aims to measure the efficiency of test models to highlight errors in a transformation. This activity is important since it provides the most adapted test set for a given transformation.

Fleurey *et al.* qualify a set of test models regarding its coverage of the input domain, the input metamodel [Fleurey 2009]. Such an approach often leads to the definition of more models than necessary. The coverage of the whole input domain is targeted, even if only a subpart of the input domain is used by the transformation. To tackle this issue, Sen *et al.* prune the metamodel to extract only the subparts involved in the transformation before providing the tests [Sen 2009].

Enhancing Test Model Set. Other approaches are proposed to qualify test. For example, Fleurey *et al.* propose an adaptation of a bacteriologic algorithm to model transformation testing [Fleurey 2004]. The bacteriologic algorithm [Baudry 2005] is designed to automatically improve the quality of a test model set. It measures the power of each data to highlight errors to (1) reject useless test models, (2) keep the best test models, (3) combine the latter to create new test models. Their adaptation of this algorithm consists in creating new test models by covering part of the input domain still not covered.

In order to produce new test model, Mottu *et al.* provide MuTest that generates multiple assertions to highlight the single error voluntary introduced in a copy of the initial program [Fraser 2010]. Once the new test model is found, the test set is minimised in order to keep the test set as small as possible.

The EvoSuite tool automatically improves a test set for the Java language [Fraser 2011]. It relies on mutation testing to produce a reduced set of assertions maximizing the mutation score. In order to produce the new test model, EvoSuite directly handles Java byte code. The tool is so dependent on the Java language that it makes its adaptation to other transformation languages particularly difficult.

Error Localisation in Model Transformation. In the context of model transformation, at that time around 2010, few approaches provide solution and tools to efficiently assist the test engineer.

Wimmer *et al.* propose a debugging support for the QVT *Relation* language [Wimmer 2009]. QVT transformations are transformed to colored Petri nets transformations in order to get a representation of the transformation execution. Existing tools based on Petri nets then allow a step by step execution of the transformation. Looking for errors is simplified, but nevertheless remains long when the transformation is complex.

Test Set Selection. Test case selection techniques seek to reduce the number of test cases. The selection is not only temporary (*i.e.,* specific to the current version of the program) but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the system under tests [Yoo 2012].

Literature (*e.g.,* [Engström 2008, Engström 2010, Ernst 2003]) recognises two types of approaches static and dynamic. The *dynamic approach* consists in executing the tests and recording the code executed during each test. This is the execution trace of a test. A test depends on a piece of code if this piece of code is in its execution trace. The *static approach* does not require to execute the tests. It relies on computing the dependency graph from the source code or some representation of it (*e.g.,* bytecode for Java). Several dependency graphs can be used ([Biswas 2011, Engström 2008, Engström 2010]): Data dependency graph, Control dependency graph, Object relation diagram, etc.

Different kinds of granularity can be considered [Engström 2010] from individual instructions (*e.g.,* [Rothermel 1993]) to modules (*e.g.,* [White 1992]) or external components (*e.g.,* [Willmor 2005]) passing through functions/methods (*e.g.,* [Elbaum 2003, Zheng 2007]) and classes (*e.g.,* [White 2005, Hsia 1997]). Using a smaller granularity gives better precision but is more costly [Engström 2010].

Summary. This state of the art shows that no existing approach provides a language independent solution to localise errors in a transformation neither to qualify test model set. Moreover, the existing approaches mostly rely on static analysis what can be a problem when the input metamodel coverage is the quality criterion to end the process.

Before implementing a test case selection approach in a major IT company, we would like to analyse among several approaches, which ones are the most adapted and why.

4.3 Contributions

4.3.1 Trace Mechanism

To solve the aforementioned issues, we defined our own trace approach [Glitia 2008]. It relies on two metamodels: the Local Trace metamodel corresponding to traceability in a single transformation and the Global Trace metamodel describing traceability in a transformation chain.

Local Trace metamodel. The Local Trace metamodel, shown in Figure 4.1, contains two main concepts: *Link* and *ElementRef* expressing that one or more elements of the source models are possibly bound to elements of the target ones. Properties and classes may be traced through respectively *PrimitivePropertyRef ClassRef*. The rule or the black-boxes (*e.g* a native library call) producing the traceability link is traced using the *RuleRef* and the *BlackBox* concepts. An *ElementRef* refers to the real element (*EObject*) of the input (resp. output) model instantiating the *ECore* metamodel. Rules and elements are gathered in containers.

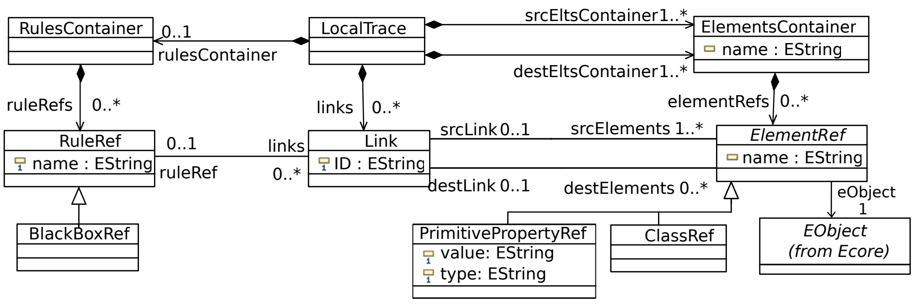


Figure 4.1: Local Trace metamodel

Global Trace metamodel. The Global Trace model [Glitia 2008] ensures the navigation from local trace models to transformed models and reciprocally as well as between transformed models. It can also be used to identify the local trace associated to a source or destination model.

Each *TraceModel* produced during a transformation and referring to a *LocalTrace*, binds two sets of *LocalModels* as shown in Figure 4.2. These are shared out by transformations, indicating that they are produced by one transformation and consumed by another.

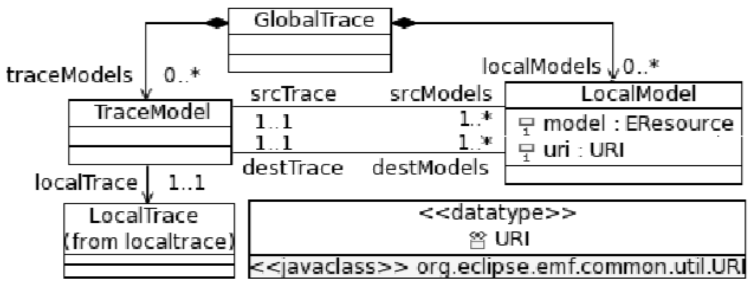


Figure 4.2: Global Trace metamodel

These two metamodels are the results of works conducted by a master student, Flori Glitia who I co-supervised with Dr Cedric Dumoulin.

4.3.2 Error localisation

Errors can be everywhere in the transformation. Their detection is easier if the search field is reduced to the faulty rule, *i.e.* the rule that creates the incorrect element (or doesn't create an expected element) in the output model. Our algorithm aims to reduce the investigation field by highlighting the rule sequences which lead to the observed error [Aranega 2009].

This algorithm supposes that either the error consists in an erroneous property (*e.g.* with an unexpected value) in an element, or an error on an element (e.g. added or missing). It looks for the faulty rule that leads to this error. In the first case, the faulty rule is easily identified. It corresponds to the *RuleRef* coupled to the *Link* associated to the *ElementRef* referring the selected element. In the second case, the faulty rule is the one which calls the last rule involved in the creation of the selected element. Causes can be a missing or misplace rule call.

We detail the algorithm in the second case:

1. select the faulty element and identify the model to which it belongs
2. from the Global Trace model, recover the Local Trace model whose the previously identified model is one of the output models
3. look for the *ElementRef* corresponding to the selected element in the local trace *destContainer*
4. recover the *RuleRef* associated to the *ElementRef*by navigating through the trace links,
5. store the *RuleRef* and the *eObject* type
6. search, in the *destContainer*, the *ElementRef* which have their *eObject* linked by an association to the *eObject* corresponding to the *ElementRef* identified in step 3
7. apply recursively the algorithm from step 3 on each element found in step 4

The recursive call stops when no direct linked *eObject* can be found in step 6. The rule is called by no other one; it is an entry point of the transformation. Technically, it is materialised by the storage of a *null* pointer. Thus, the algorithm results in a kind of tree representing the successions of rules producing the selected element. Fixing the error localised in the transformation requires the identification of the input model elements leading to this incorrect output element.

Finally, due to the non exhaustiveness of tests and the complexity of building oracles, test of a single transformation can be missed at the expense of test of the whole transformation chains. For this purpose, we developed a new version of the algorithm adapted to transformation chains. Not only the successive rules are stored but also any element of the input model that was useful to the creation of the faulty output element. The algorithm is then again applied on each of these elements. The final result is a set of rules corresponding to the set of potential faulty rules on the whole transformation chain.

The two versions of the algorithm were applied with success in Gaspard on transformations written with different transformation languages (QVTO and a Java API) in the context of Vincent Aranega's thesis that I co-supervised with Professor Jean-Luc Dekeyser.

4.3.3 Mutation Analysis and Model Transformations

Mutation analysis aims to qualify a test set and to enhance it until reaching a satisfactory threshold preliminary fixed. Mutation analysis relies on the following assumption: if a test set can reveal the faults voluntary and systematically injected in various versions of the program under test, then this set is able to detect involuntary faults. For this purpose, variations of the original program to test, called *mutants*, are created by applying a mutation operator that injects an error. Each mutant contains a single error. Each mutant is run with each test data. A mutant is considered *killed* if its execution with at least one test data is different from the original program execution with the same test data. If no test data highlight the error injected in the mutant, it is considered *alive*. New test data are created to kill more mutants until the satisfactory threshold is reached [DeMillo 1978].

The mutation analysis process is divided into four parts: *(i)* creating the mutants and the original test set, *(ii)* executing the original program and the mutants with each test data, *(iii)* computing the mutation score (*i.e.* the ratio of mutants killed by the test data set compared to the whole mutants) and *(iv)* producing new test data when the mutation score is too low. The three first activities can be automated. Our contribution concerns the last one in the context of model transformations.

Our proposal is based on the following hypothesis: building a new test model from scratch can be extremely complex, while taking advantage of existing test models could help to construct new ones. Consequently, we developed an approach helping to create new test models from modifications of relevant existing models.

The approach is composed of two major steps as shown in Figure 4.3: (i) the selection of a relevant pair $(test\ model, mutant)$ (activity 1) and (ii) the creation of a new test model by adequately modifying the one identified (activity 2).

Selection of Relevant Pairs (Model, Mutant). This step relies on an intensive use of the model transformation traceability briefly presented in section 4.3.1. The proposed approach relies on the assumption that test models owning elements which are used by the *mutated rule* (and so by the mutated instruction, *i.e.,* the rule and respectively the instruction containing the error voluntary injected in the program under test) are good candidates to be improved to kill the mutant. Indeed, this rule has been executed on elements of these test models, but the resulting models do not differ from the ones of the original transformation executions possibly because of neutralisation by the remainder of the transformation. The mutant is the

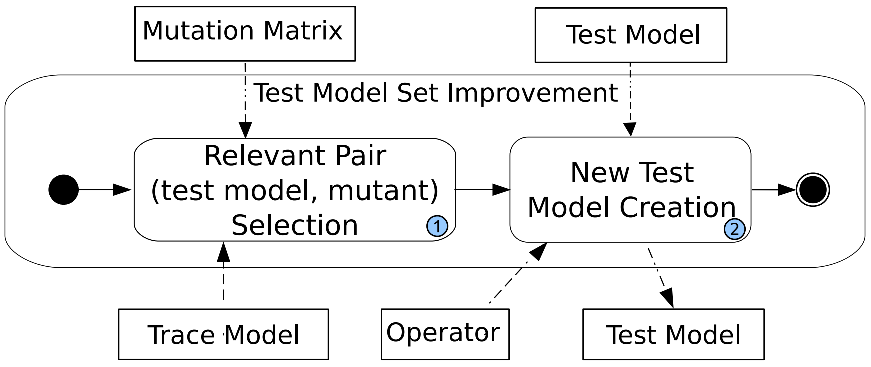


Figure 4.3: Test Improvement Process Overview

exact copy of the original transformation except the mutated instruction. The difference of the outputs can only result from the execution of this mutated instruction and thus of the mutated rule. The results of the mutants execution for each data test are stored in a matrix, named mutation matrix. Thus, this mutation matrix specifies if a mutant has been killed by a given data test or not. The traceability mechanism and the mutation matrix gathering all the traces allow the tester to identify these candidate models and, for each one, to highlight the elements consumed and produced by the mutated rule [Aranega 2011] .

Modelling Mutation Operators. In the context of model transformations, mutation operators were either language specific [Tisi 2009, Fraternali 2009] or only specified through a text description [Mottu 2006]. However, such an informal specification does not enable any reasoning, or treatment automation. To solve these two issues and provide a generic definition independent of any transformation language but dedicated to model transformation, we provide a metamodel representation of the mutation operators defined in [Mottu 2006].

The *original contribution* consists in modeling mutation operators based on their effects upon the data manipulated by the transformation under test instead of based on their implementation in the transformation language being used. Each mutation operator is designed as a metamodel expressing how the operator may be applied on any transformation [Aranega 2014b]. The 10 mutation operators defined by Mottu *et al.* [Mottu 2006] lead to the creation of 10 mutation operator metamodels.

Example of the Relation to the Same Class Change Operator (RSCC) mutation operator. "The RSCC operator replaces the navigation of one reference towards a class with the navigation of another reference to the same class." [Mottu 2006] The RSCC operator can be applied on the input or the output metamodel of the transformation but only if it exists, in the metamodel, at least two EReferences between the two same EClasses. One EReference is originally navigated, the other is navigated by a mutant. Thus applied on a model transformation, RSCC operator replaces the original navigation by another to the same EClass.

The RSCC operator metamodel is presented in Figure 4.4. In order to ensure its independence from any transformation and transformation language, it is expressed on generic concepts that can appear in any transformation whatever the used language. Indeed, the RSCC operator metamodel uses the EMOF metamodel (on the left of Figure 4.4) to specify the input or output elements of the transformation the operator is applied on. The abstract classes *Navigation* and *Replacement* were introduced to factorise references and increase reusability between operator metamodels as shown in the Annex [Aranega 2013]. *RSCC* class corresponds to the mutation operator. *initNavigation* is the EReference initially navigated by the transformation whereas *newNavigation* is the EReference navigated after the mutation. Additional constraints are necessary to ensure the viability of the mutant created. The first constraint prevents the mutants to be equivalent. The second constraint requires the two EReferences being to the same EClass.

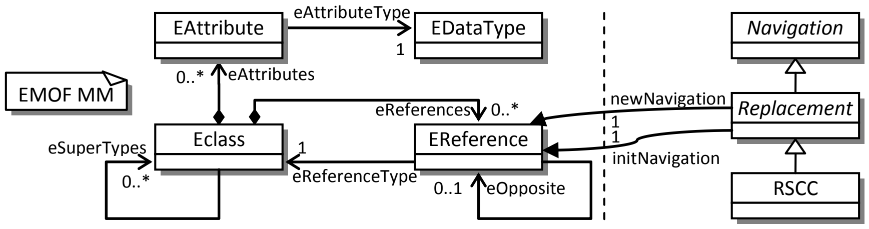


Figure 4.4: *RSCC* operator metamodel

The application of one mutation operator on one transformation returns *mutation models* which conform to the corresponding mutation operator metamodel. Whereas a mutation metamodel is generic, its models are dedicated to one model transformation, but still language independent. Those mutation models are based on the input and output metamodels of the transformation. They define how input/output model elements could be treated by the original transformation and how the mutants would treat them.

Creation of a New Test Model by Modifying an Existing One based on Pattern Identification. Based on the abstract representations of the operators and their definition on the input or output metamodel of the transformation, it is possible to identify why a mutant remains alive, and give some recommendations to modify existing test models that in their new versions should kill the mutant. For each mutation operator, few test model *patterns* (*i.e.* specific configurations in the model) leaving a mutant alive are identified. For each pattern, modifications that should kill the mutant are provided.

It has to be noticed that these patterns and their recommendations available

[Aranega 2014b, Aranega 2013] are specified at a meta level based only on the abstract representation of the operators. The proposed approach provides an automatic analysis of the situation and advises some first modifications to be performed; in a lot of cases, they will be enough to kill the mutant.

Researches around mutation analysis lead by Vincent Aranega during his PhD that I co-supervised with Professor Jean-Luc Dekeyser. They also result from collaborations with Dr Jean-Marie Mottu that I first supervised as a post-doc and with Dr Benoît Baudry.

4.3.4 Test Set Selection after a Change in the Program

If all the results presented above in this chapter were applied on model transformations, the following ones concern more traditional programs.

Tests are crucial for Worldline, a major IT company, for different reasons. First, the company provides payment and transactional-services that are critical to its customers. Errors, bugs or denial of service are not allowed. Second, it provides solutions from design to deployment and maintenance. However, running all the tests on a project may take hours because they require installing and configuring database or another environment as well as testing abnormal running conditions such as timeout on server connection. In a daily development process, developers can not run the tests after a change to check the impacts of their modifications. Since they have no tool to detect tests impacted by a change, they very often skip tests during the day and these only run at night thanks to continuous integration servers.

We decided with Worldline to improve this situation. We work closely with a transversal team in this company that provides tools, expertise and support to the development teams. This team is aware of the issues met by the field teams and look for adapted solutions to simplify developers work while guaranteeing quality. To convince upper management of possibly imposing a change in work practices of thousands of developers, the transversal team needs convincing hard data on the pros and cons of the technique it will propose. We report in this chapter some conclusions on our first studies.

While experimenting with some existing test case selection tools, we were confronted with different issues. Most of the projects of the company are written in Java, we therefore limited ourselves to this language or at least to the Object-Oriented paradigm.

Classification of Issues. Problems in test selection approaches arise when there is a break in the dependency graph representing the system. Such breaks may occur for several reasons. In our case, we identified four categories of reason. Note that this list might not be exhaustive.

Third-party breaks: The application uses external libraries or frameworks for which the source code is not available. In this case, a static analysis of the code cannot trace dependencies through the third-party code execution.

Multi-program breaks: The application consists in several co-operating programs (*e.g.,* client/server application). In this case, an analysis focused on one single program cannot trace dependencies into the other program.

Dynamic breaks: The application contains execution of code generated on-the-fly (*e.g.,* using the language reflective API). In this case, an analysis of the source code cannot yield the dependencies that will occur at execution.

Polymorphism breaks: The application uses polymorphism. In this case, a dependency analysis may reach a class on which nobody else depends because all dependencies point to a superclass of it.

Experimental Setup. This experiment aims to identify tests methods impacted by a change in a method of the application, and, does not deal with change detection. Each method of the application is considered as arbitrarily changed and the ability for the approach to detect a test covering the changed method is studied.

According to the experiment protocol, the dynamic approach is set as the oracle. We are looking for the answers to the following research questions. In the context of Wordline projects, what is the impact on the results of a chosen granularity (RQ1), the third-party breaks issue (RQ2), the dynamic breaks issue (RQ3), the polymorphism breaks issue (RQ4), combining the solutions to different problems (RQ5), changing the same methods repeated times (as occurs in real life) (RQ6) and considering real commits (that change several methods jointly) (RQ7) on test case selection? All experiment follow the same pattern:

i. We fixed one version of the source code on which we work. This version never changes, all changes are virtual. Test coverage is given by the dynamic approach (Jacoco tool[1] [Lingampally 2007]) that is our baseline.
ii. We consider as "changed" each method of the application that is covered by at least one test. Considering one static approach, we try to recover the test cases covering this method.
iii. From the test cases recovered, we compute different metrics: Precision, Recall and F-Measure.
iv. The metrics values are averaged over all Java methods (covered by at least one test) to produce a result for the static approach considered.
v. The same process is repeated for another static approach and we compare their respective results to answer the research question. The difference in the results is considered as the impact of the problem that one of the two static approaches solves.

[1]http://eclemma.org/jacoco/

To answer RQ1, we apply the same static approach twice, once on all methods (Steps ii and iv), and once on all classes (Steps ii and iv, replacing “methods” by “classes”).

To answer RQ2, we apply one static tool (Infinitest[2]) that can overcome the third party break issue and another one (Moose[3] [Ducasse 2000]) that cannot.

To answer RQ3, RQ4, and RQ5, we apply the same static tool (Moose) including or not the solutions to the different problems considered. For RQ5 (combining all solution), we will not be able to include the solution to the third party break issue, because it cannot be easily done with Moose.

To answer RQ6, we use a weighted mean where each method has a weight corresponding to the number of commits (in the history of the system) where it appears in.

Finally for RQ7, we apply the same static tools (Infinitest and Moose) on all methods in one case and all system commits in the other case. Commits differ from individual methods in that they may change many methods (up to 125 in one case) potentially covered by the same tests.

Projects. To perform our experiments, we selected three projects (P1, P2 and P3). P1 and P2 are financial applications with more than 400 KLOC. P1 is a service (in term of Service Oriented Architecture (SOA)) dealing with card management. P2 is an issuing banking system based on SOA and reusing the card management system developed in P1 (P2 uses P1 as a third party). P3 has no relation with the two other projects, and is an e-commerce application. P2 and P3 test suites are mainly composed of integration tests, that ensure the good behaviour of the application with its dependencies and the data base. P1 test suite includes mainly unit tests that guarantee the results of the algorithms. In these projects, each test is a Java method using JUnit[4].

P1, P2 and P3 are big applications (hundred of KLOC). P1 includes 5,323 valid tests; P2, 168; and P3, 3,035. In P1, the tests cover 4,720 methods (48%), in P2, only 3,261 methods (6%), and in P3, 8,143 methods (18%).

Test execution (compilation and test execution included) requires 3 hours for P1; 180 minutes for P2; and 30 minutes for P3. This only includes the tests that we are considering in our experiments, not all tests of the projects. This time is mainly due to the setup of each test (database population, server startup and configuration); test data volume; and the fact that there are abnormal conditions tests (timeout). Commits seem rather big: over hundred methods, 18 files.

[2]http://infinitest.github.io/
[3]http://www.moosetechnology.org/
[4]http://junit.org/

Results. Table 4.1 gives the metrics of the results of each static approach (by comparison to the dynamic approach).

RQ1 – Class vs Method Granularity. To answer this Research Question, we consider Moose approaches at method and class granularity. For P1 and P3, the method granularity has a clear advantage with higher *Precision* and much less *Selected tests*. P2 behaves as expected: more *Selected tests* would usually result in lower *Precision*. The three projects have better *Recall* at class granularity (resp. 50%, 37%; and 19%). Again this is conform to expectations as more *Selected tests* would usually result in higher *Recall*. Note that the *Precision* and *Recall* results do not seem very good overall. This could be due to the different issues identified previously and that are not treated in this experiment.

RQ2 – Third-Party Breaks impact. Because Infinitest works at the class level and considers some dependencies (*e.g.* references to the classes), we do the same for Moose. Only the third party break is not bypassed in the "Moose for Infinitest" approach. *Recall* is better for Infinitest for the three projects (resp. 72%, 66%, and 44%) which is normal since it selects more tests. The difference however is small. *F-Measure* is more consistent and gives better results for the three projects to Moose (not solving the third party breaks). Based on the *F-Measure* results, one could conclude that there is no urgent need to solve the third party issue on our three projects.

RQ3 – Dynamic Breaks Impact. For the three projects, we see almost no change between Moose solving one of the specific Dynamic Break issues and Moose not solving any issue. The only exception is slightly more *Selected tests* for P2 when solving the Attribute Automatic Initialisation issue, followed by significantly better *Precision*, *Recall* and consequently *F-Measure*. The first conclusion would be that it is mostly useless to try to solve these issues. We will see however by analyzing results of RQ5 that issues may be intertwined and that solving one alone might not be enough.

RQ4 – Polymorphism Breaks Impact. The three projects have more *Selected tests*. *Precision* improves significantly for P2 and P3 and remains equal for P1. Again an improvement here is unexpected since we selected more tests. *Recall* improves dramatically for P1, from 36% to 91% and significantly for P2 and P3. And of course *F-Measure* improves also for the three projects. The conclusion is that it was very important to solve this specific issue in our cases. P1 particularly shows excellent results, with >90% *Recall*, a still good *Precision* (43%, about half of the selected tests do cover the changed method) and a similarly good rate of *Selected tests*.

RQ5 – Impact of Combining Solutions. The combination of all implemented solutions gives very good results. Overall, the results for P1 and P3 are similar to the ones of the previous experiment and P2 is showing more *Selected tests*, much better *Precision*, *Recall*, and *F-Measure*. Another conclusion is that by answering all the issues (minus the third party breaks that Moose cannot solve easily), we end up with very good results, *Precision* ranges from 34% (P3) to 61% (P2), and *Recall* ranges from 41% (P3) to 91% (P1). These results position the static approach as a viable solution to the test case selection problem.

RQ6 – Weighting of Results with the Number of Commits. This new experiment consistently brings marginal decrease in *Precision* and *Recall* and small increase in *Selected tests*. All these results points towards a worsening of the results. This would suggest that the methods where static approaches are not able to find the tests are more frequently committed. This is not good news, but the differences are small (typically one percentage point) and would need to be more formally tested in a specific experiment.

RQ7 – Aggregation of the Results by Commit. P1 and P3 improve their *Precision* (resp. from 43% to 55%; and from 34% to 49%), but P2 decreased (from 61% to 45%). So good news for P1 and P3, that are more selective but also more precise. Being more selective, the *Recall* results were bound to worsen: the approaches select less tests than they should according to our baseline. This is what happens with P1 and P2 (resp. from 91% to 81%; and from 64% to 45%), but P3, which previously had the lower *Recall*, improved it (from 41% to 56%). This is coherent with P3 also improving its *Precision*.

Overall Conclusions on the Test Case Selection Experiment. Three overall conclusions can be drawn from these experiments. First, problems might be intertwined and one needs to combine several solutions together to fully resolve any of the issues. Second, problems do not have the same impact on the projects. This might be the consequence of different coding conventions or rules. For example the attribute initialisation issue is not present in P1. Such issues might be helped by establishing better coding conventions. Third, considering commits instead of individual Java methods tend to worsen the results with approaches that are too selective to keep the same level of good results. The large size of the commits might be an important factor in this behaviour. As already stated, a positive consequence of the entire experiment (still in progress) would be to see developers make smaller commits that would help in test case selection and would also give them better and faster feedback on their changes.

Another conclusion from the issue classification would be that even in one category, issues can unfortunately be very different and require each a specific solution.

Table 4.1: Comparison of the static approaches to the dynamic one for test case selection considering all Java methods individually

		Selected Tests			Precision			Recall			F-Measure		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
Jacoco (dynamic)		0.8%	1%	0.4%	-	-	-	-	-	-	-	-	-
Infinitest	RQ2	23%	5%	3%	9%	39%	15%	72%	66%	44%	12%	43%	18%
Moose for Infinitest		19%	2,2%	2%	10%	27%	18%	70%	63%	41%	13%	44%	20%
Moose (classes)	RQ1	8%	1%	0.7%	15%	23%	13%	50%	37%	19%	18%	26%	12%
Moose (methods)		0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ delayed exec.		0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ anonym. classes	RQ3	0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ attributes		0.4%	0.2%	0.1%	43%	17%	24%	36%	17%	13%	35%	17%	15%
Moose w/ polymorphism	RQ4	3%	0.4%	2%	43%	25%	34%	91%	26%	41%	50%	25%	29%
Moose w/ att. & anon. & polym. & delayed exec.	RQ5	3%	0.8%	2%	43%	61%	34%	91%	64%	41%	50%	62%	29%
Moose combining issues & weighted with nb of commits	RQ6	3%	1%	2%	42%	59%	33%	92%	62%	39%	50%	60%	28%
Moose combining issues & considering commits	RQ7	4%	3%	6%	55%	45%	49%	81%	45%	56%	56%	40%	42%

For our experiments, we used the dynamic approach as oracle because it is safe and accurate. However, this approach has two major drawbacks: First, this approach is not generic; it depends strongly from the data used for the tests. Second, if a test is failing, no execution trace is recorded and it cannot be selected by this approach.

Researches around test selection after a change in a code were lead by Vincent Blondeau in the context of his CIFRE PhD with Worldline, that I co-supervised with Dr Nicolas Anquetil.

4.4 Perspectives

Testing Transformation Chain. As discussed in Chapter 2, with localized transformations, the sizes of the transformation decrease whereas their number in a chain increases. We have proposed in the present chapter solutions to detect error in a single transformation or to automate the qualification of data test set. New approaches have to be defined to tackle the test of transformation chain similarly to integration test in traditional programming. Our error localization algorithm has been extended for chains. However, since only the input model is known by users, a test non-satisfied by a transformation has to be translated backwards along a model transformation chain into an equivalent constraint over the input language of the chain. Richa recently tackled this issue [Richa 2015] but lots of related challenges remain unsolved.

Debugging the Resulting Application. In this chapter, we only tackled the test of model transformations (or chains). Once the transformation and the chain will be tested enough to be trustworthy, new approaches to debug the resulting generated application should be proposed. Errors in this type of applications will result from a bad design in the input models. The idea is to tune the input model and to analyse the consequences on the resulting application. Or in the opposite to identify what is expected in the application and to consequently modify the input model. We started research in this domain [Aranega 2014a] but lots of tracks remain unexplored.

Automating Qualification of Data Test Set for Traditional Programming Paradigm. Mutation operators dedicated to model transformation were designed with metamodel. Moreover, patterns and recommendations were provided in order to automate the introduction of new test models. Such an approach based on meta modeling should certainly be adapted to traditional languages.

Selecting Tests after Change in the Code. Some identified issues cannot be solved with a static or dynamic approach alone. Hybrid solutions should be experimented. Moreover, such experiments on real changes have to be performed. In parallel, we want to study if such test selection approaches have an impact on engineers way to test their application. For this purpose, a first study is performed to understand how often engineers are testing the application, in which context (*e.g.* before committing) which tests and so on. Same study will be then performed while test selection approaches will be integrated in the partner company.

Adapting Tests after Change in the Code. Selecting tests after change in the code enables the developers to get a quick feed back on the performed evolution. Such an approach allows them to focus on a smaller set of test and no more on the full test set. However, the changes performed on the source code may break some tests that must consequently be adapted. Consequently, tests are important for the success of the software and its evolution, while paradoxically they are also a serious burden during evolution, because they need to be maintained as well. Adapting tests after source code evolution is not an easy task, and software engineers need tools and methods that help to assess the nature of the relationship between these two artefacts [Zaidman 2011].

CHAPTER 5

Co-evolution Supported by Metamodels

5.1 Problems

According to Lehman's law, a software system must evolve, or it becomes progressively less useful [Lehman 1980]. Evolving software does not only mean considering the source code. As already seen in this document, software can rely on model, metamodel and transformation. In that case, each of these artefacts may evolve with consequences on the others. This multi-dimensionality present in MDE also exists in traditional programming. The previous chapter quickly sketches the issue of evolving the tests after source code evolution. Other artefacts such as requirements, documentation, database, librairies may be impacted [Mens 2005b]. Such a mechanism is called *co-evolution*.

In biology, co-evolution occurs when changes in at least two species reciprocally affect each other's evolution. In software engineering, "*the necessity to achieve co-evolution between different types of software artifacts or different representations of them*" has been considered as a research challenge [Mens 2005b]. Modification in one representation should always be reflected by corresponding changes in other related ones to ensure consistency of all involved software artifacts.

Co-evolution aims to preserve an interdependence between artefacts. Strictly speaking co-evolution involves simultaneity between the artefact evolutions. In practice, *Migration* occurred more often. It corresponds to the development activity in which artefacts are updated in response of the evolution of a first one to re-establish the interdependence.

Several problems need to be solved when tackling co-evolution or migration. First the interdependence between the artefacts should be precisely defined. Then simultaneous or in response evolution has to be performed, according rules, strategies or operators that need to be defined.

This chapter deals with metamodel-transformation co-evolution and database schema-program co-evolution.

5.2 Previous State of the Art

Different Types of Co-Evolution. Lämmel identifies four types of what he calls *coupled transformation* [Lämmel 2004]. Figure 5.1 sketches them.

- *no reconciliation*, the evolution of the artefacts is known to be restricted such that the artefact is changed without challenging the interdependence relation, then the interdependent artefacts can be kept as is.
- *degenerated reconciliation* occurs when one of the artefact is, for example, generated from another. Then the evolution of this latter can be easily derived from the other by generating again.
- *symmetric reconciliation* implies that the interdependent artefacts evolve simultaneously within a same "transformation" that has two twin versions one for each artefact.
- *asymmetric reconciliation* relies on the assumption that the history (or in fact the evolution) of one artefact can be derived from the evolution of theother artefact. When this latter evolves it is thus possible to migrate the former.

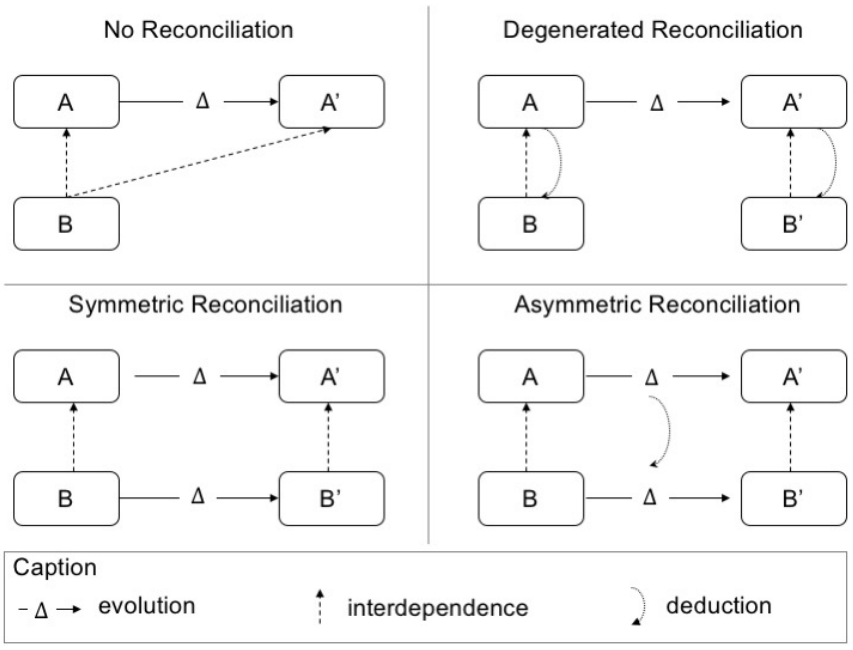


Figure 5.1: Four different types of coupled transformation

We extended this classification to introduce three new types (Figure 5.2) [Etien 2005]:

- *independence*, the different artefacts independently evolve, potentially impacting the interdependence. This latter is eventually checked, for example with metrics, to be re-established.
- *dependence* is a subclass of asymmetric reconciliation when it is always the same artefact that master the migration of the other. It is the most frequent case of asymmetric reconciliation.
- *double-dependence* is also a subclass of asymmetric reconciliation. It corresponds to the case where each artefact can master at its turn the migration.

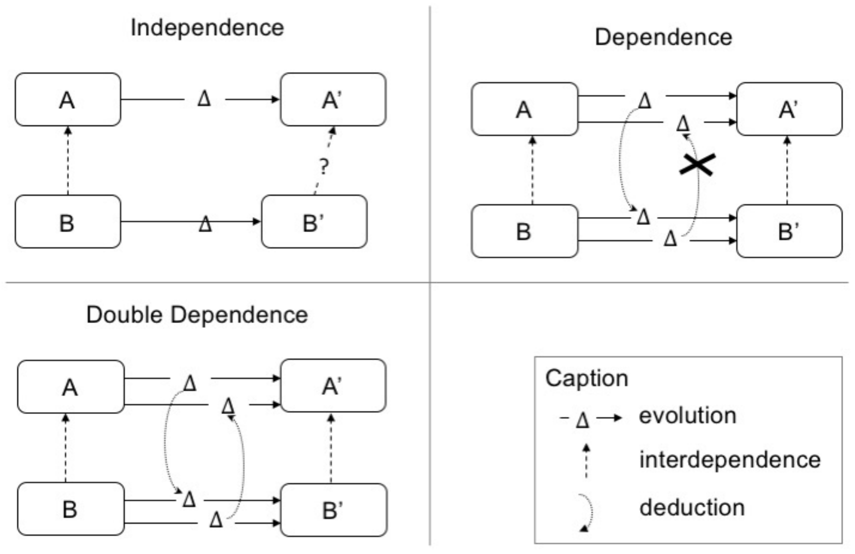


Figure 5.2: Three additional cases of co-evolution

Model-Metamodel Co-evolution. Model-metamodel co-evolution is a case of *dependence* where the evolution of the metamodel may break the conformance of the existing models.

Rose *et al.* propose a classification of model migration approaches [Rose 2009]. This classification highlights three ways to identify needed model updates: *(i)* manually, *(ii)* operator-based, and *(iii)* by inference. In manual approaches, migrations are defined by hand specifically to a given system [Rose 2010a]. In operator based, the metamodel changes are defined in terms of co-evolutionary operators [Herrmannsdoerfer 2009]. Those operators define conjointly the evolution on the metamodel and its repercussions on the models. Finally, in inference approaches, versions of metamodels are compared and differences between them are used to semi-automatically infer a transformation that expresses model updates [Garcés 2009, Cicchetti 2008].

Transformation-Metamodel Co-evolution. Similarly to the previous case, transformation-metamodel co-evolution is a case of *dependence* where the evolution of the metamodel may break the interdependence with the transformations.

Steel *et al.* are aware that metamodel evolution might imply transformation migration [Steel 2007]. However, they do not propose a transformation migration approach. They define model typing that establishes the metamodel evolution spectrum in which a transformation remains valid.

Roser *et al.* present an approach that enables semi-automatic transformation migration after metamodel evolution [Roser 2008]. However, they only support the cases in that the source and target metamodels belong to the same knowledge area represented by a reference ontology. The approach is thus not generic and adaptable to any transformation whatever the input and output metamodels.

Database Artefacts Co-evolution. Türker exposes the possible schema evolutions in SQL99 [Türker 2001]. In practice, a referenced artefact can not be removed since this evolution impacts the artefacts referencing it; only evolution with no impact on the existing artefacts (*e.g.* table, view, procedure) are allowed. Thus, for example, it is possible to remove a domain only if it is not referenced in any of other entities. Database administrators have to manually ensure, before evolution, that it is possible.

Nagy *et al.* present a static technique for identifying the exact source code location in an object-oriented program from where a given SQL query was sent to the database [Nagy 2015]. This query extraction approach is a first step for impact analysis. Maule *et al.* analyze the impact of database schema changes on external object oriented program [Maule 2008]. The only entity types they take into account are tables and columns; views, stocked procedures or triggers for example are not handled. Database entities are used in a program only through queries that can be built dynamically. They increase string analysis precision and reduce the parts of the program to which the analysis is applied with program slicing.

After having identified all possible atomic changes, Qiu *et al.* analyze the real impacts caused by these atomic schema changes by mining a project's version control history [Qiu 2013]. They use a database application's co-change history to estimate the application code area affected by a schema change. Co-changes inside the database schema are not taken into account.

Summary. There exists different types of co-evolution according to which entity evolves and how it is propagated to the other. However, each type relies on a strong relationship between the two co-evolving entities. In the context of model driven engineering, model-metamodel co-evolution has been widely studied, whereas transformation-metamodel co-evolution is only considered an issue without real solutions where the metamodel evolution may invalidate the transformation. In the context of traditional programs using database, co-evolution is recognised as an issue. However, co-changes inside the database schema are not studied.

5.3 Contributions

5.3.1 Transformation-Metamodel Co-evolution

Transformation-Metamodel Interdependence. We name the interdependence between transformations and metamodels *domain conformance* [Mendez 2010] and defined it as follows.

A precondition of transformations is that input models conform to source metamodel. The transformation has to guarantee that the generated output models conform to target metamodel. As a consequence, the relationship between a transformation and its source metamodel is different from the relationship between the transformation and its target metamodel. We call those relationships *domain* and *codomain* respectively and define domain conformance in terms of them.

The domain of a transformation is the set of elements involved in the source patterns (also called left hand side parts) of the transformation. Similarly, the codomain of a transformation is the set of elements involved in the target patterns (also called right hand side parts) of the transformation. There is an additional restriction: models produced by the transformation should conform target metamodel.

A transformation T and its input and output metamodels (resp. MM_{in} and MM_{out}) respect the domain conformance relationship if the domain of T is a submetamodel[1] of MM_{in}, if the codomain of T is a submetamodel of MM_{out} and if all the well-formedness constraints defined in MM_{out} concerning concepts present in both MM_{out} and the codomain of T also exist in the codomain of T.

Transformation Migration. Transformation migration can be split into three phases: 1) impact detection, 2) impact analysis, and 3) transformation adaptation. During impact detection, the transformation inconsistencies caused by metamodel evolution are identified *i.e.* where transformation does not satisfy domain conformance. During impact analysis, the set of transformation updates needed to re-establish domain conformance is obtained possibly by using human assistance. Finally, during transformation adaptation, updates found in step two are applied.

In order to provide a solution independent of any platform and language, our approach relies on an abstract representation of the transformation also called Platform Independent Transformation (PIT) [Bézivin 2003], and a change metamodel. The change metamodel gathers all the changes that can occur on the source or

[1]in the sense of typing defined by Steel *et al.* in [Steel 2007]

target metamodel elements such as `Modify Multiplicity` or `Eliminate Inheritance`

Impact analysis phase aims to identify the transformation updates needed to re-establish domain conformance. Our proposal relies on potential modifications. The current supported set is presented below. According to the kind of change applied on the source or target metamodel elements and defined in the change metamodel, suggestions to update the transformation are proposed. Note that some changes may need human intervention. Suggestions for a first set of changes occurring on the metamodels associated to the transformation are listed below and were defined in [Mendez 2010].

- **Rename Class/Property:** All the occurrences of the renamed class/ property should be updated in the PIT model by changing its name.
- **Move Property:** If this change occurs in the *source metamodel*, the path to access the property should be updated in all statements involving it. If the change occurs in the *target metamodel*, it is necessary to move the statement in the rule creating the new owner of the property and the path to access the other elements involved in the statement have to be updated.

  If *property is moved between classes that are in the same transformation pattern*, then the path of the property should be updated.
- **Modify Property:** No suggestions in this case because it is impossible *a priori* to ensure the typing of property. Furthermore, in case of multiplicity, the management of a unique element may become a collection. The impact should be resolved manually.
- **Introduce Class:** Introducing a class in the *source metamodel* does not affect domain conformance but metamodel coverage of transformation. Hence, the propagation of this change depends of the purpose of the transformation and cannot be automated.

  Introducing a class in the *target metamodel* affects domain conformance only if this class is mandatory (*i.e.* referred with multiplicities “1”, “1..*”, “1..n”). This change cannot be automatically propagated. Hence, the user should do it manually by choosing one of two options: 1) write a new transformation rule that creates elements conform to the new class based on concepts of source metamodel and modify an existing rule to call the former. 2) modify the target pattern of an existing transformation rule (mostly the one creating the owner of the added class) by including the new class on it.
- **Introduce Property:** The same analysis than class addition can be done except that statement and not rule are added. This statement should be either in the class containing the property or in its subclasses. If the property is mandatory (*i.e.* multiplicity: "1", "1..*", or "1..n") this action has to be done.
- **Replace Class:** All the occurrences of the replaced class should be changed by the new one. All the statements involving the properties of the replaced

class should be fixed manually by finding its equivalence in the new one.

- **Eliminate Class:** If the removed class belonged to the *target metamodel* and was the root of a pattern, the corresponding rule has to be removed. Otherwise, all the statements using the class or its properties should be removed. If after that elimination some patterns become empty, the corresponding transformation rule should be eliminated.
- **Eliminate Property:** All the statements involving the removed property should be removed.
- **Introduce Inheritance:** The same analysis than introduce property can be done except that the mandatory inherited properties have to be included to the transformation rules creating the subclasses if they not already exist in the rule creating the super class.
- **Eliminate Inheritance:** All the statements that involves inherited properties by subclasses should be removed.

Researches around Transformation-Metamodel co-evolution took place in the context of David Mendèz' internship, a master student that I supervised and a collaboration with Professor Rubby Casallas from Universitad de los Andes, Bogota, Columbia.

5.3.2 Database and Program Co-Evolution.

An information system relies on a database gathering the data and a set of programs treating these data. Two major approaches are adopted concerning the separation between the database and the programs. Initially, data treatments were implemented as stocked procedures in the database management system. Such an approach ensures the homogeneity of the data treatment whatever the applications using the database. Due for example to the difficulty to debug SQL or stored procedure code or to the necessity for the developers to learn a new language, treatments were externalised in the application. However, information systems are no more a single application on top of a database. Nowadays, they correspond to several applications with different technologies and languages on top of the database. Such new database usages lead to clone code treatment between applications, increase maintenance difficulty due for example to dynamic request, or inconsistencies between database and program. To centralise treatments, ensure data coherence and increase performances, information system engineers go back to stocked procedures usage and use advanced functionality like views or triggers.

During the last two years, based on the case study of the information system of the CRIStAL laboratory, I have been working on the co-evolution of the database structure and the database treatment embedded in the database management system. DB structure refers to *tables*, *columns* and integrity constraints. DB program corresponds to code internally executed in the database system management and relies on for example *views*, *functions* and *triggers* In this section, the co-evolution between the database structure and external program for example written in Php, Java or Pharo is not tackled.

Even if the data structure and the treatment are gathered into the database management system, their co-evolution is not an easy task. None of the existing commercial tools that we tested helps in the co-evolution of entities belonging to the structure or the program part of a same database schema[2]. Some of them enable to represent only tables and their dependencies (*e.g.* PgAdmin[3] or Navicat[4]) other deal also with views (*e.g.* Toad[5] or Maestro[6]) but none of them deal with functions that are only handled as strings.

Interdependence between Database Structure and Program. The relations between database entities of the structure and the program internal to the database management system are classical Access/Reference or Invocation links. The complexity of the interdependence does not come under its type but under the different use cases of each entity by the others. All the links are gathered in a metamodel shown in Figure 5.3. Dark grey classes belong to the FAMIX metamodel. They are extended to build the SQL metamodel. Eight grey classes correspond to the concepts composing the DB structure. White classes describe the DB internal program. An *Expression* is either a request, an invocation to a function or a column. A *Request* may content a select clause, a where clause, a from clause and other clauses (*e.g.* order by or having). Each of these clauses are expressions. A request can also refer tables in the from clause. A *Function* contains expressions. Consequently, a *Column* can be used by a *View*, a *Function* or another *Column*. This later case corresponds to the primary - foreign key link and belongs to the DB structure. The other cases correspond to link between structure and program entities. A *Function* can be called by a *View*, a *Trigger*, an other *Function* or in a *Column* definition, column belonging to a view (*i.e.,* in the select clause of the request composing the view). Finally, a *Table* can be used either in a *Function* or a *View* definition in the from clause of a request. According to the database management system other entities are concerned by these links such as datatypes, sequences or aggregates for PostgreSQL. To simplify, we do not represent them in the metamodel.

Impact Analysis between Database Structure and Program. Based on this SQL/PL/pgSQL metamodel, it is possible to identify when an entity is used and in the other way round which entities it uses. For this purpose, we developed an API to query, from a given entity, all the dependent entities and all the used ones. This

[2]Database Schema = DB structure + DB internal program.

[3]http://www.pgadmin.org

[4]http://www.navicat.com

[5]http://www.toadworld.com

[6]https://www.sqlmaestro.com

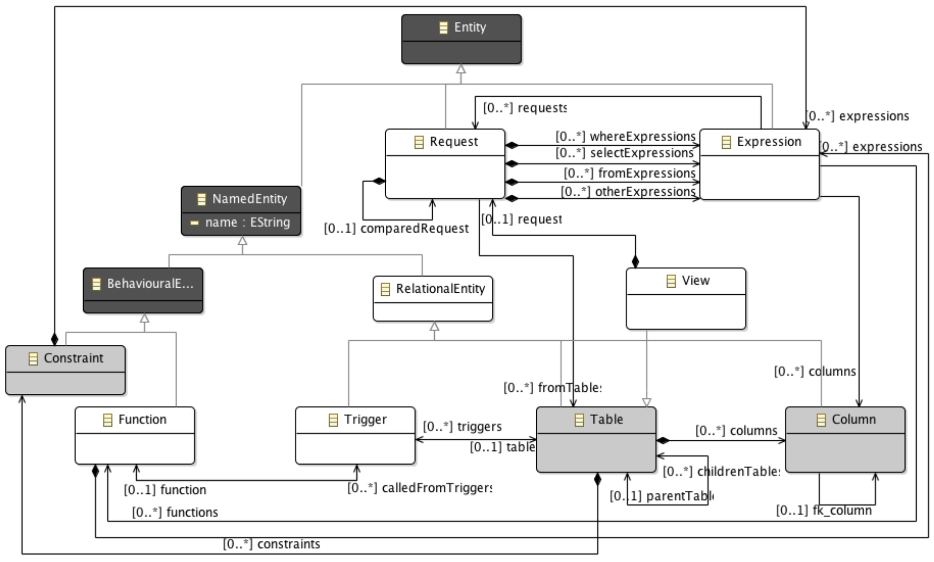


Figure 5.3: SQL/PL/pgSQL Metamodel

API is independent of the entity type (*e.g.* Table, View, Function). This API allows us to identify all the impacted entities when a given entity is changed.

The information system of CRIStAL, among others, manages the members of the laboratory according to the team they belong to, the financial support paying them and keeps in memory the whole history. Thus, for example, the database registers that I was in the Dart team as an Inria Post doc from September 2006 to August 2007. Then I was associate professor of normal class of Computer science in the same team from September 2007 until September 2012. Currently and since October 2012, I am associate professor in the RMod team. This concrete example shows that the database deals for example with *person*, *support*, *team*, *rank* (grade in French), *employer*. Based on this information for each member of the laboratory, thanks to functions and views, it is for example possible to foresee retirements by computing the ages pyramid or support ending.

Figure 5.4 illustrates an example of visualisation enabling the database administrator to very quickly identify all the entities (*e.g.* tables, views, triggers and functions) that may be impacted if the *personne* table (in the centre of the butterfly visualisation) is changed. It is possible *(i)* to work at the column level in order to have an impact analysis at a finer grain than at table level as presented in the figure and *(ii)* to recursively apply the API in order to have in cascade impacts. Renaming or removing a column in this central table may directly impact around fifty entities (depending on the column), and even more by taking into account transitive impact. Indeed, in PostgreSQL, view structure can not be modified if other views depend on it. It has to be dropped and then create again in its new version. Manually studying such impacts in real case studies is quickly error prone and tedious.

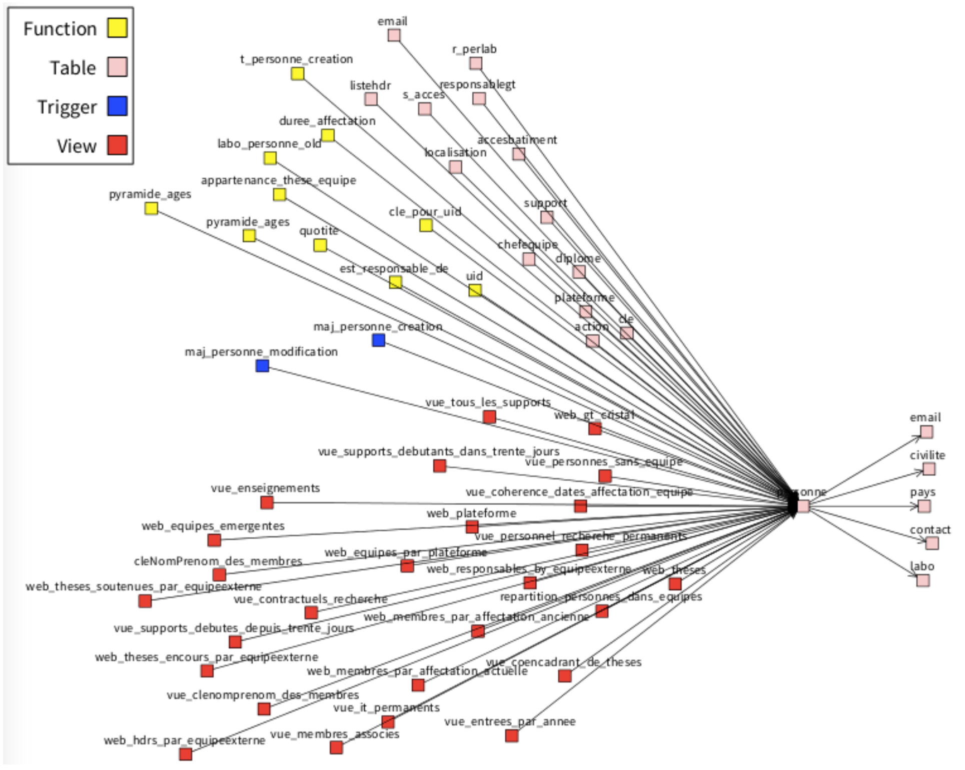

Function 🟨
Table ▨
Trigger 🟦
View 🟥
email
r_perlab
t_personne_creation
listehdr
s_acces
responsablegt
duree_affectation
labo_personne_old
accesbatiment
localisation
appartenance_these_equipe
pyramide_ages
support
cle_pour_uid
pyramide_ages
quotite
chefequipe
diplome
est_responsable_de
uid
plateforme
cle
action
maj_personne_creation
maj_personne_modification
vue_tous_les_supports
web_gt_cristal
vue_supports_debutants_dans_trente_jours
vue_personnes_sans_equipe
vue_enseignements
vue_coherence_dates_affectation_equipe
personne
email
civilite
pays
contact
labo
web_plateforme
web_equipes_emergentes
vue_personnel_recherche_permanents
web_equipes_par_plateforme
cleNomPrenom_des_membres
web_responsables_by_equipeexterne
web_theses
web_theses_soutenues_par_equipeexterne
vue_contractuels_recherche
repartition_personnes_dans_equipes
web_membres_par_affectation_ancienne
vue_supports_debutes_depuis_trente_jours
web_theses_encours_par_equipeexterne
vue_coencadrant_de_theses
web_membres_par_affectation_actuelle
vue_clenomprenom_des_membres
vue_it_permanents
vue_entrees_par_annee
web_hdrs_par_equipeexterne
vue_membres_associes

Figure 5.4: All the entities using the *personne* table and which it uses

Moreover, we distinguish in the metamodel the different clauses of the request. The impacts won't be the same according that the changed element appears in one or the other clause. For example, removing a column appearing in the select clause of request may be managed differently if this column is the only one appearing in the select clause, if the request is used with a comparison operator like UNION or EXCEPT, or if the select clause refers several columns. In the two first cases, error can be raised. Similar analysis are possible for each type of changes occurring on each DB structure type and for each relationship between entities. This analysis corresponds to the one we did concerning transformation-metamodel co-evolution. It is part of future work.

This work on database structure and program co-evolution is in progress. It relies on a close collaboration with Olivier Auverlot, the architect of the CRIStAL information system.

5.4 Perspectives

Transformation-Metamodel Co-evolution. Results concerning transformation-metamodel co-evolution were preliminary but precursory. Definitions of the changes possibly occurring on the metamodels and their associated suggestions need to be further detailed. Moreover, the implementation of the suggested impacted changes remained a future work. Using a high order transformation, operators, or specific strategy defined by the user were contemplated.

Even if research was undertaken on this topic, it seems that a real transformation-metamodel co-evolution engine tackling models scalability is still missing. It should also be relevant to study the consequences of localised transformation usage on transformation-metamodel co-evolution.

Unifying model-metamodel & transformation-metamodel co-evolution. Given the similarities between model-metamodel and transformation-metamodel co-evolution [Rose 2010b], it may be possible to use a single tool to manage both types of co-evolution. Several potential advantages are apparent for a unified co-evolution approach. Firstly, the metamodel evolution is expressed only once and enables the co-evolution of both models and transformations. Secondly, implementation of co-evolution tools may be simplified via the use of localised transformations. Thirdly, a unified approach may provide a foundation for supporting other types of co-evolution, such as model-model (more commonly termed model synchronisation). Given these advantages, future research should assess the extent to which co-evolution approaches can be unified, and compare unified and specialised approaches to establish their strengths and weaknesses.

Semi-Automatic database and program co-evolution. The last step of the database structure and program co-evolution is the migration of the program after a change in the structure. For this purpose, we first plan to provide migration recommendations, to the database architect, for each impacted entities. These recommendations rely on the type of the involved entities, the link between them and the initial change occurring on the entity. In a first time, they will be provided as plain text. Adding, removing or renaming a *Column* for example will not have the same impacts. Possible changes on each entity type will be listed as a set of operators for example by using the approach, we proposed [Rolland 2004]. In a second time, we plan to provide SQL code corresponding to the recommendations in order to perform the migration and resulting in a consistent and valid schema.

Database and DBMS external program co-evolution. The database structure and program co-evolution tackled in this chapter only concerns program embedded in the Data Base Management System (DBMS) and more specifically functions written in PL/pgSQL or SQL. The state of the art has shown that others are providing solutions for database structure and external program[7] co-evolution. They only take into account the structure (*i.e. Tables* or *Columns*). We would like to enhance these approaches by enabling the co-evolution when any schema entity *i.e.* also *Views* and *Functions* are used by external programs. Moreover, the existing approaches often provide solutions when the program is written in Java. In CRIStAL information system, Pharo programs are using database entities. We would like to provide an approach independent as much as possible from the language used in the program, by relying on metamodels.

[7]outside the DBMS

CHAPTER 6

Conclusion and Perspectives

This chapter first provides a summary of the contributions reported in the manuscript and then proposes short-term and long-term perspectives for the research in this area.

6.1 Main Results

The results reported in this document aim to design systems of good quality and easily maintainable and then maintain them. Two types of systems were studied: traditional programs and model transformation chains. Four phases of a system lifecycle are analysed: design, architecture modifications, test and co-evolution.

Designing Model Transformation Chains. Model transformation chains were monolithic and difficult to reuse and maintain. Based on our experience in the Gaspard environnement we provided a new way to design model transformations and chains: *the localised transformations*. Each transformation definition relies on small metamodels gathering only the concepts useful to it. The `Extend` operator enables the execution of the transformation on models conforming to larger metamodels. The in-place mechanism has been extended for transformation with different input and output metamodels. This new way to design model transformations has consequences on chains. Chain design is no more metamodel centric but transformation centric. In other words, chains have to be specified in terms of successive steps to perform and no more in terms of intermediary states to reach. Localised transformations enable the definition of transformation libraries. We provided mechanisms based on feature models to choose the adapted transformations according to user's requirements. New chaining constraints were also defined. The Gaspard environment was completely redesigned with localised transformations. In the same domain, adding a new chain from existing one for example to target a new language is easy and quick since lots of transformations can be reused. This work is independent of the language used to specify the transformations. It results from collaborations. It has been published in international journals and conferences and transferred to industry.

Software Architecture Modifications. Among the three different forms of software architecture modifications: *refactoring, renovating and rearchitecting*, we provide concepts, methods and tools for two of them (refactoring and rearchitecting). Rearchitecting is performed to enhance software quality. We developed a graphical tool handling several possible alternative versions of the future architecture and enabling the developer to validate constraints on each of them. Consequently, with such a tool the better version can be chosen according to the way each version validates the constraints.

Repetitive sequences of operator changes were observed in the context of architecture refactoring or rearchitecting. These sequences slightly differ for each application. We introduced the notion of transformation pattern and we provided a tool to play and record once the sequence, to automatically configure it and play it again in another context. The proposed approach is independent of the used programming language (patterns were observed in Java and PHARO programs). It relies on program and change operator abstractions. This work in the context of a PhD thesis and an international collaboration. It was the topic of several international publications.

Testing Supported by Metamodels. Model transformations are used to generate source code from models. Once defined, they are executed on multiple inputs. They have to be trustworthy to be sure that when an error is discovered in the generated code, it results from an error in the input model and not one in the transformation. Thus, as any other program, but also for this reason, model transformations have to be tested. Existing tools and approaches for traditional programs cannot be used as such and need to be adapted. We provided a traceability mechanism based on two metamodels respectively dedicated to transformations and chains. This mechanism enables error localizations when outputs do not comply with the expected ones. It has also been used in a mutation approach to qualify test set and create new test models when required. Once again these results are generic and independent of the used transformation language.

When the tests are numerous, running all of them after a change in the program may be very long, sometimes several hours. Consequently, the feedback to developers is no more immediate, tests are only run at night. We are currently working with a major IT company to study the impact of the test case selection on the way developers use tests. We compared static and dynamic approaches on three industrial projects. We identified some issues related to object-oriented paradigms and classified them. This work on tests was conducted in the context of two PhD thesis and lead to industrial or academic collaborations.

Co-evolution Supported by Metamodels. Systems, in case of traditional program or model transformation chains, are composed of several interdependent artefacts. When one of them evolves, others have to be consequently migrated to maintain the interdependence. We studied co-evolution in the context of metamodel and transformation and also in the context of database schema and internal program. For this purpose, we have clearly specified the relationship between metamodel and transformation. We also defined the relationships between Pg/PLSQL concepts. Recommendations to perform on the transformation after changes in the input or the output metamodels are provided. Such recommendations for the database context do not yet exist but are part of future work. Currently, it is already possible to analyse changes on elements of the database or the internal program and to identify their consequences on the internal program elements. Metamodel-transformation co-evolution has been generically treated by a master student whereas database schema-program is jointly studied with the administrator of the CRIStAL information system.

6.2 Perspectives

Definition of chains from on-the-shelves transformations. Designing transformation chains remains difficult and long. Transformations and chains are often built from scratch. Localised transformations have thrown the basis for the definition of chains from on the shelves transformations. However, even if the proposed approach is independent of used transformation language, we observed that it is domain dependent. Introduction of genericity in transformation could help to reuse transformations from one domain to another.

Moreover, a strong knowledge of the domain is required to choose the needed transformations and build the chains. Chaining constraints are taken into account, but non functional requirements should also be managed. This implies to provide a language to express them, to check them and to mix them with already handled structural constrains.

Finally, without a real tool enabling the developer to tune the input model directly according to performance or execution expected on the generated code, transformation chains won't be more used in industry. Feedbacks from the code on the model are far to be immediate.

Tool Supported Architecture Modifications. Even with the tool we provided, architecture modifications are essentially manually performed directly on the code. They do not take benefit from models, contrarily to design. Or, in the opposite, they are undertaken at a very high level without real connection to the code. Architecture modifications require taking a step back and consequently, abstraction is useful. At the end, code is expected and since some pieces already exist, they have to be reused. These two abstraction levels should be accessible on demand. Moreover, architecture modifications are often done with a specific idea in mind, *e.g.,* removing cycle, breaking large classes or decreasing coupling. It would be good to be able to check if the goal is achieved as soon as possible and possibly try several alternatives before choosing the best one. Some bricks already exist or were proposed in this manuscrit. Nevertheless a complete approach is still missing.

Tool Supporting Co-evolution in the Large. Systems are composed of several artefacts. Lehman has taught us that a software system must evolve, or it becomes progressively less useful. We have seen that these evolutions may break interdependences between the artefacts. Co-evolution is a solution to this issue. However, it is currently mostly handled between two artefacts only. We provided examples on metamodels and transformations and on database schema and programs. Thus, evolutions on the database schema lead for example to migrations of the program. But, these changes in the program may impact tests and documentations that will need to evolve in cascade. Biologists distinguish co-evolution by pair where only two species jointly evolve to diffuse co-evolution, when there are more species. In software engineering, only this former type of co-evolution is managed whereas the other type is observed. Before jointly managing several artefacts, perhaps will it be necessary to easily adapt existing co-evolution mechanisms whatever the involved artefacts.

Bibliography

[Alanen 2008] M. Alanen and I. Porres. *A metamodeling language supporting subset and union properties*. Software and System Modeling, vol. 7, no. 1, pages 103–124, 2008.

[Aranega 2009] Vincent Aranega, Jean-Marie Mottu, Anne Etien and Jean-Luc Dekeyser. *Traceability Mechanism for Error Localization in Model Transformation*. In ICSOFT, Bulgaria, July 2009.

[Aranega 2011] Vincent Aranega, Jean-Marie Mottu, Anne Etien and Jean-Luc Dekeyser. *Using Trace to Situate Errors in Model Transformations*. In José Cordeiro, AlpeshKumar Ranchordas and Boris Shishkov, editeurs, Software and Data Technologies, volume 50 of *Communications in Computer and Information Science*, pages 137–149. Springer Berlin Heidelberg, 2011.

[Aranega 2012] Vincent Aranega, Anne Etien and Sebastien Mosser. *Using Feature Model to Build Model Transformation Chains*. In Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems, MODELS'12, pages 562–578, Berlin, Heidelberg, 2012. Springer-Verlag.

[Aranega 2013] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry and Jean-Luc Dekeyser. *Annexe and Experimentation material*. https://sites.google.com/site/mutationtesttransfo/, 2013.

[Aranega 2014a] Vincent Aranega, Antonio Wendell De Oliveira Rodrigues, Anne Etien, Frédéric Guyomarch and Jean-Luc Dekeyser. *Integrating Profiling into MDE Compilers*. International Journal of Software Engineering & Applications (IJSEA), vol. 5, no. 4, page 20, July 2014.

[Aranega 2014b] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry and Jean-Luc Dekeyser. *Towards an Automation of the Mutation Analysis Dedicated to Model Transformation*. Software Testing, Verification and Reliability, vol. 25, no. 5-7, pages 653–683, August-November 2014.

[Avgeriou 2013] Paris Avgeriou, Michael Stal and Rich Hilliard. *Architecture Sustainability*. IEEE Software, vol. 30, no. 6, pages 40–44, 2013.

[Baroni 2014] Alessandro Baroni, Henry Muccini, Ivano Malavolta and Eoin Woods. *Architecture Description Leveraging Model Driven Engineering and Semantic Wikis*. In 11th Conference on Software Architecture, pages 251–254, 2014.

[Baudry 2005] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Yves Le Traon. *From Genetic to Bacteriological Algorithms for Mutation-Based Testing*. STVR Journal, vol. 15, no. 2, pages 73–96, June 2005.

[Beck 2004] Kent Beck and Cynthia Andres. Extreme programming explained: Embrace change (2nd edition). Addison-Wesley Professional, 2004.

[Bézivin 2003] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois and Damien Pollet. *Reflective Model Driven Engineering*. In G. Booch P. Stevens J. Whittle, editeur, Proceedings of UML 2003, volume 2863 of *LNCS*, pages 175–189, San Francisco, October 2003. Springer.

[Biswas 2011] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy and Srihari Sukumaran. *Regression Test Selection Techniques: A Survey*. Informatica (03505596), vol. 35, no. 3, 2011.

[Budgen 2003] David Budgen. Software design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 édition, 2003.

[Cicchetti 2008] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio. *Automating Co-evolution in MDE*. In Proc. EDOC, pages 222–231. IEEE Computer Society, 2008.

[Cordy 2009] Jim Cordy. *Eating our own Dog Food: DSLs for Generative and Transformational Engineering*. In GPCE, 2009.

[Cuadrado 2011] Jesús Sánchez Cuadrado, Esther Guerra and Juan de Lara. *Generic Model Transformations: Write Once, Reuse Everywhere*. In Jordi Cabot and Eelco Visser, editeurs, ICMT, International Conference on Theory and Practice of Model Transformations, volume 6707 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2011.

[Czarnecki 2003] Krzysztof Czarnecki and Simon Helsen. *Classification of Model Transformation Approaches*. Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, 2003.

[de Lara 2012] Juan de Lara and Esther Guerra. *Reusable Graph Transformation Templates*. In AGTIVE, volume 7233 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2012.

[DeMillo 1978] R. A. DeMillo, R. J. Lipton and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer*. Computer, vol. 11, no. 4, pages 34–41, April 1978.

[dic] *Definition of Pattern by Merriam-Webster dictionary*. http://www.merriam-webster.com/dictionary/pattern. Accessed: 2015-06-30.

[Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET '00, June 2000.

[Ducasse 2011] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval and Tudor Girba. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Research report, Inria, 2011.

[Ehrig 2009] Karsten Ehrig, Jochen M. Küster and Gabriele Taentzer. *Generating instance models from meta models*. Software & Systems Modeling, vol. 8, no. 4, pages 479–500, 2009.

[Elbaum 2003] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel and S. Kanduri. *Understanding the effects of changes on the cost-effectiveness of regression testing techniques*. Journal of Software Testing, Verification, and Reliability, 2003.

[Engström 2008] Emelie Engström, Mats Skoglund and Per Runeson. *Empirical evaluations of regression test selection techniques: a systematic review*. In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pages 22–31. ACM, 2008.

[Engström 2010] Emelie Engström, Per Runeson and Mats Skoglund. *A systematic review on regression test selection techniques*. Information and Software Technology, vol. 52, no. 1, pages 14–30, 2010.

[Ernst 2003] Michael D Ernst. *Static and dynamic analysis: Synergy and duality*. In WODA 2003: ICSE Workshop on Dynamic Analysis, pages 24–27. Citeseer, 2003.

[Etien 2005] A. Etien and C. Salinesi. *Managing requirements in a co-evolution context*. In Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, pages 125–134, Aug 2005.

[Etien 2010] A. Etien, A. Muller, T. Legrand and X. Blanc. *Combining Independent Model Transformations*. In Proceedings of the ACM SAC, Software Engineering Track, pages pp. 2239–2345, 2010.

[Etien 2012] Anne Etien, Vincent Aranega, Xavier Blanc and Richard F. Paige. *Chaining Model Transformations*. In Proceedings of the First Workshop on the Analysis of Model Transformations, AMT '12, pages 9–14, New York, NY, USA, 2012. ACM.

[Etien 2015] Anne Etien, Alexis Muller, Thomas Legrand and RichardF. Paige. *Localized model transformations for building large-scale transformations*. Software & Systems Modeling, vol. 14, no. 3, pages 1189–1213, 2015.

[Fleurey 2004] F. Fleurey, J. Steel and B. Baudry. *Validation in model-driven engineering: testing model transformations*. In Proceedings of MoDeVa, pages 29–40, Nov. 2004.

[Fleurey 2009] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller and Yves Le Traon. *Qualifying input test data for model transformations*. Software and System Modeling, vol. 8, no. 2, 2009.

[Fluri 2007] Beat Fluri, Michael Wuersch, Martin PInzger and Harald Gall. *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction*. IEEE Transactions on Software Engineering, vol. 33, no. 11, pages 725–743, 2007.

[Fluri 2008] Beat Fluri, Emanuel Giger and Harald Gall. *Discovering Patterns of Change Types*. In 23rd International Conference on Automated Software Engineering, pages 463–466, 2008.

[Fowler 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

[France 2003] Robert France, Sudipto Ghosh, Eunjee Song and Dae-Kyoo Kim. *A Metamodeling Approach to Pattern-Based Model Refactoring*. IEEE Software, vol. 20, no. 5, pages 52–58, 2003.

[Fraser 2010] Gordon Fraser and Andreas Zeller. *Mutation-driven generation of unit tests and oracles*. In Proceedings of the 19th international symposium on Software testing and analysis, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[Fraser 2011] Gordon Fraser and Andrea Arcuri. *EvoSuite: automatic test suite generation for object-oriented software*. In ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT FSE, 2011.

[Fraternali 2009] Piero Fraternali and Massimo Tisi. *Mutation Analysis for Model Transformations in ATL*. In International Workshop on Model Transformation with ATL, Nantes, France, June 2009.

[Gamatié 2011] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet and Jean-Luc Dekeyser. *A Model-Driven Design Framework for Massively Parallel Embedded Systems*. ACM Trans. Embed. Comput. Syst., vol. 10, no. 4, pages 39:1–39:36, November 2011.

[Garcés 2009] K. Garcés, F. Jouault, P. Cointe and J. Bézivin. *Managing Model Adaptation by Precise Detection of Metamodel Changes*. In Proc. ECMDA-FA, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.

[Glitia 2008] Flori Glitia, Anne Etien and Cedric Dumoulin. *Traceability for an MDE Approach of Embedded System Conception*. In ECMDA Traceability Workshop, Germany, 2008.

[Guerra 2012] Esther Guerra. *Specification-Driven Test Generation for Model Transformations*. In Zhenjiang Hu and Juan de Lara, editeurs, ICMT, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2012.

[Hemel 2008] Zef Hemel, Lennart C. L. Kats and Eelco Visser. *Code Generation by Model Transformation*. In Proceedings of the 1st international conference on Theory and Practice of Model Transformations, ICMT'08, pages 183–198, Berlin, Heidelberg, 2008. Springer-Verlag.

[Herrmannsdoerfer 2009] M. Herrmannsdoerfer, S. Benz and E. Juergens. *COPE - Automating Coupled Evolution of Metamodels and Models*. In Proc. ECOOP, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[Hsia 1997] Pei Hsia, Xiaolin Li, David Chenho Kung, Chih-Tung Hsu, Liang Li, Yasufumi Toyoshima and Cris Chen. *A technique for the selective revalidation of OO software*. Journal of Software Maintenance: Research and Practice, vol. 9, no. 4, pages 217–233, 1997.

[ISO 2006] ISO. *International Standard – ISO/IEC 14764 IEEE Std 14764-2006*. Rapport technique, ISO, 2006.

[Jackson 2002] Daniel Jackson. *Alloy: A New Technology for Software Modelling*. In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '02, 2002.

[Javed 2012] Muhammad Javed, Yalemisew Abgaz and Claus Pahl. *Composite Ontology Change Operators and their Customizable Evolution Strategies*. In Workshop on Knowledge Evolution and Ontology Dynamics, collocated at 11th International Semantic Web Conference, pages 1–12, 2012.

[Jiang 2015] Qingtao Jiang, Xin Peng, Hai Wang, Zhenchang Xing and Wenyun Zhao. *Summarizing Evolutionary Trajectory by Grouping and Aggregating Relevant Code Changes*. In 22nd International Conference on Software Analysis, Evolution, and Reengineering, pages 1–10, 2015.

[Kim 2013] Miryung Kim, David Notkin, Dan Grossman and Gary Wilson Jr. *Identifying and Summarizing Systematic Code Changes via Rule Inference*. IEEE Transactions on Software Engineering, vol. 39, no. 1, pages 45–62, 2013.

[Küster 2006] Jochen M. Küster and Mohamed Abd-El-Razik. *Validation of model transformations: first experiences using a white box approach*. In

Proceedings of the 2006 international conference on Models in software engineering, 2006.

[Lämmel 2004] R. Lämmel. *Coupled Software Transformations*. In First International Workshop on Software Evolution Transformations, November 2004.

[Lano 2013] Kevin Lano and Shekoufeh Kolahdouz Rahimi. *Optimising Model-transformations using Design Patterns.* In 1st International Conference on Model-Driven Engineering and Software Development, pages 77–82, 2013.

[Lehman 1980] Meir M. Lehman. *Programs, life cycles, and laws of software evolution.* Proc. IEEE, vol. 68, no. 9, pages 1060–1076, September 1980.

[Lingampally 2007] R. Lingampally, A. Gupta and P. Jalote. *A Multipurpose Code Coverage Tool for Java.* In System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, pages 261b–261b, jan 2007.

[Maule 2008] Andy Maule, Wolfgang Emmerich and David S. Rosenblum. *Impact Analysis of Database Schema Changes.* In In Proceedings of International Conference on Software Engineering (ICSE'08, pages 451–460, 2008.

[Mendez 2010] David Mendez, Anne Etien, Alexis Muller and Rubby Casallas. *Towards Transformation Migration After Metamodel Evolution.* In Model and Evolution Wokshop, Olso, Norway, October 2010.

[Mens 2005a] T. Mens, G. Taentzer and O. Runge. *Detecting Structural Refactoring Conflicts Using Critical Pair Analysis.* Electronic Notes in Theoretical Computer Science, vol. 127, no. 3, pages 113–128, April 2005.

[Mens 2005b] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld and M. Jazayeri. *Challenges in software evolution.* In Principles of Software Evolution, Eighth International Workshop on, pages 13–22, Sept 2005.

[Mens 2005c] Tom Mens, Krzysztof Czarnecki and Pieter Van Gorp. *A Taxonomy of Model Transformations.* In Language Engineering for Model-Driven Software Development, 2005.

[Mens 2007] Tom Mens, Gabriele Taentzer and Olga Runge. *Analysing Refactoring Dependencies Using Graph Transformation.* Software and Systems Modeling, vol. 6, no. 3, pages 269–285, 2007.

[Mikkonen 1998] Tommi Mikkonen. *Formalizing Design Patterns.* In 20th International Conference on Software Engineering, pages 115–124, 1998.

[Mottu 2006] Jean-Marie Mottu, Benoit Baudry and Yves Le Traon. *Mutation Analysis Testing for Model Transformations*. In ECMDA 06, Spain, July 2006.

[Murphy-Hill 2009] Emerson Murphy-Hill, Chris Parnin and Andrew P. Black. *How We Refactor, and How We Know It*. In 31st International Conference on Software Engineering, pages 287–297, 2009.

[Nagy 2015] Csaba Nagy, Loup Meurice and Anthony Cleve. *Where was this SQL query executed? a static concept location approach*. In Yann-Gaël Guéhéneuc, Bram Adams and Alexander Serebrenik, editeurs, Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering, pages 580–584. IEEE, 2015.

[Negara 2013] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson and Danny Dig. *A Comparative Study of Manual and Automated Refactorings*. In 27th European Conference on Object-Oriented Programming, pages 552–576, 2013.

[Nguyen 2010] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi and Tien N. Nguyen. *Recurring Bug Fixes in Object-oriented Programs*. In 32nd International Conference on Software Engineering, pages 315–324, 2010.

[Nguyen 2013] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, T.N. Nguyen and H. Rajan. *A study of repetitiveness of code changes in software evolution*. In 28th International Conference on Automated Software Engineering, pages 180–190, 2013.

[Oldevik 2005] Jon Oldevik. *Transformation Composition Modelling Framework*. In Proceedings of the Distributed Applications and Interoperable Systems Conference, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer, 2005.

[Olsen 2006] G. Olsen, J. Aagedal and J. Oldevik. *Aspects of Reusable Model Transformations*. In Proceedings of the ECMDA Composition of Model Transformations Workshop, pages pp. 21–26, 2006.

[Pigoski 1997] T. Pigoski. Practical software maintenance. best practices managing your software investment. John Wiley and Sons, 1997.

[Pilgrim 2008] Jens Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach and Yolande Berbers. *Constructing and Visualizing Transformation Chains*. In ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture, pages 17–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[Qiu 2013] Dong Qiu, Bixin Li and Zhendong Su. *An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications*. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 125–135, New York, NY, USA, 2013. ACM.

[Ray 2012] Baishakhi Ray and Miryung Kim. *A Case Study of Cross-system Porting in Forked Projects*. In 20th International Symposium on the Foundations of Software Engineering, pages 1–11, 2012.

[Richa 2015] Elie Richa. *Qualification of Source Code Generators in the Avionics Domain : Automated Testing of Model Transformation Chains*. PhD thesis, TELECOM ParisTech, 2015.

[Rivera 2009] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista and Antonio Vallecillo. *Orchestrating ATL Model Transformations*. In Proc. of MtATL 2009, pages 34–46, Nantes, France, July 2009.

[Rolland 2004] Colette Rolland, Camille Salinesi and Anne Etien. *Eliciting gaps in requirements change.* Requir. Eng., vol. 9, no. 1, pages 1–15, 2004.

[Rose 2009] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A. C. Polack. *An Analysis of Approaches to Model Migration*. In Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems, October 2009.

[Rose 2010a] L.M. Rose, D.S. Kolovos, R.F. Paige and F.A.C. Polack. *Model Migration with Epsilon Flock*. In Laurence Tratt and Martin Gogolla, editeurs, Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

[Rose 2010b] Louis Rose, Anne Etien, David Mendez, Dimitrios Kolovos, Fiona Polack and Richard F. Paige. *Comparing Model-Metamodel and Transformation-Metamodel Co-evolution*. In Model and Evolution Wokshop, Olso, Norway, October 2010.

[Roser 2008] Stephan Roser and Bernhard Bauer. *Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space*. Journal on Data Semantics XI, pages 32–64, 2008.

[Rothermel 1993] Gregg Rothermel and Mary Jean Harrold. *A Safe, Efficient Algorithm for Regression Test Selection*. In Proceedings of the International Conference on Software Maintenance (ICSM '93), pages 358–367. IEEE, September 1993.

[Sanchez Cuadrado 2008] J. Sanchez Cuadrado and J. Garcia Molina. *Approaches for Model Transformation Reuse: Factorization and Composition*. In Proceedings of the International Conference on Model Transformation, volume 5063 of *LNCS*, pages pp. 168–182. Springer-Verlag, 2008.

[Santos 2015a] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Marco Tulio Valente. *OrionPlanning: Improving modularization and checking consistency on software architecture*. In 3rd IEEE Working Conference on Software Visualization, VISSOFT 2015, Bremen, Germany, September 27-28, 2015, pages 190–194. IEEE, 2015.

[Santos 2015b] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse and Marco Tulio Valente. *Recording and Replaying System Specific, Source Code Transformations*. In 15th International Working Conference on Source Code Analysis and Manipulation, pages 221–230, 2015.

[Santos 2015c] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse and Marco Tulio Valente. *System Specific, Source Code Transformations*. In 31st International Conference on Software Maintenance and Evolution, pages 221–230, 2015.

[Seacord 2003] R.C. Seacord, D.A. PLAKOSH and G.A.A. LEWIS. Modernizing legacy systems: Software technologies, engineering processes, and business practices. The SEI Series in Software Engineering. Addison Wesley Publishing Company Incorporated, 2003.

[Sen 2007] Sagar Sen, Benoit Baudry and Doina Precup. *Partial Model Completion in Model Driven Engineering using Constraint Logic Programming*. In International Conference on the Applications of Declarative Programming, 2007.

[Sen 2008] Sagar Sen, Benoit Baudry and Jean-Marie Mottu. *On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing*. In ICST., Norway, April 2008.

[Sen 2009] Sagar Sen, Naouel Moha, Benoit Baudry and Jean-Marc Jézéquel. *Meta-model Pruning*. In Andy Schürr and Bran Selic, editeurs, Model Driven Engineering Languages and Systems, MODELS, volume 5795 of *Lecture Notes in Computer Science*. Springer, 2009.

[Sen 2012] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry and Jean-Marc Jézéquel. *Reusable model transformations*. Software and System Modeling, vol. 11, no. 1, pages 111–125, 2012.

[Steel 2007] Jim Steel and Jean-Marc Jézéquel. *On Model Typing*. Journal of Software and Systems Modeling (SoSyM), vol. 6, no. 4, pages 401–414, December 2007.

[Terra 2012] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki and Roberto S. Bigonha. *Recommending Refactorings to Reverse Software Architecture Erosion*. In 16th European Conference on Software Maintenance and Reengineering, pages 335–340, 2012.

[That 2012] Minh Tu Ton That, S. Sadou and F. Oquendo. *Using Architectural Patterns to Define Architectural Decisions*. In Conference on Software Architecture and European Conference on Software Architecture, pages 196–200, 2012.

[Tisi 2009] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri and Jean Bézivin. *On the Use of Higher-Order Model Transformations*. In Model Driven Architecture - Foundations and Applications, 5th European Conference, 2009.

[Türker 2001] Can Türker. *Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS*. In Selected Papers from the 9th International Workshop on Foundations of Models and Languages for Data and Objects, Database Schema Evolution and Meta-Modeling, FoMLaDO/DEMM 2000, pages 1–32, London, UK, UK, 2001. Springer-Verlag.

[Vakilian 2013] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara and Ralph E. Johnson. *A Compositional Paradigm of Automating Refactorings*. In 27th European Conference on Object-Oriented Programming, pages 527–551, 2013.

[van Amstel 2008] M. F. van Amstel, C. F. J. Lange and M. G. J. van den Brand. *Metrics for Analyzing the Quality of Model Transformations*. In 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering, 2008.

[Vanhoof 2005] Bert Vanhoof and Yolande Berbers. *Breaking up the transformation chain*. In Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005, San Diego, California, USA, 2005.

[Wagelaar 2009] Dennis Wagelaar, Ragnhild Van Der Straeten and Dirk Deridder. *Module superimposition: a composition technique for rule-based model transformation languages*. Software and Systems Modeling, 2009. Online First.

[White 1992] L.J. White and H.K.N. Leung. *A firewall concept for both control-flow and data-flow in regression integration testing*. In Software Maintenance, 1992. Proceedings., Conference on, pages 262–271, nov 1992.

[White 2005] L. White, K. Jaber and B. Robinson. *Utilization of extended firewall for object-oriented regression testing*. In Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pages 695–698, sep 2005.

[Willmor 2005] D. Willmor and S.M. Embury. *A safe regression test selection technique for database-driven applications*. In Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pages 421–430, sep 2005.

[Wimmer 2009] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Gerti Kappel, Werner Retschitzegger and Wieland Schwinger. *Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets*. In MoDELS, USA, 2009.

[Xanthakis 2000] S. Xanthakis, P. Régnier and C. Karapoulios. Le test des logiciels. Études et logiciels informatiques. Hermes Science Publications, 2000.

[Ying 2004] Annie T. T. Ying, Gail C. Murphy, Raymond Ng and Mark C. Chu-Carroll. *Predicting Source Code Changes by Mining Change History*. IEEE Transactions on Software Engineering, vol. 30, no. 9, pages 574–586, 2004.

[Yoo 2012] S. Yoo and M. Harman. *Regression testing minimization, selection and prioritization: a survey*. Software Testing, Verification and Reliability, vol. 22, no. 2, pages 67–120, 2012.

[Zaidman 2011] Andy Zaidman, Bart Van Rompaey, Arie van Deursen and Serge Demeyer. *Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining*. Empirical Software Engineering, vol. 16, no. 3, pages 325–364, 2011.

[Zheng 2007] J. Zheng, L. Williams, B. Robinson and K. Smiley. *Regression Test Selection for Black-box Dynamic Link Library Components*. In Incorporating COTS Software into Software Systems: Tools and Techniques, 2007. IWICSS '07. Second International Workshop on, may 2007.