Multiple-objectives architecture optimization by composition of model transformations

Smail Rahmoun

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0004

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

présentée et soutenue publiquement par

Smail RAHMOUN

07 Février 2017

Multiple-objectives architecture optimization by composition of model transformations

Optimisation multi-objectifs d'architectures par composition de transformation de modèles

Directeur de thèse : **M. Laurent Pautet**
Co-encadrement de la thèse : **M. Etienne Borde**

Jury

M. Eric GOUBAULT, Professeur, École Polytechnique, France	Président du jury
M. Jean-Michel BRUEL, Professeur, Université de Toulouse, France	Rapporteur
M. Xavier BLANC, Professeur, Université de Bordeaux, France	Rapporteur
M. Sébastien GÉRARD, Chargé de Recherche, CEA de Paris, France	Examinateur
M. Jan CARLSON, Maître de conférences, Université de Mälardalens, Suède	Examinateur
M. Fabrice KORDON, Professeur, Université Pierre et Marie Curie, France	Examinateur
M. Laurent PAUTET, Professeur, TELECOM ParisTech, France	Directeur de thèse
M. Etienne BORDE, Maître de conférences, TELECOM ParisTech, France	Co-encadrant de thèse

THÈSE

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Remerciements

Je tiens à remercier mes encadrants de thèse Laurent Pautet et Etienne Borde, qui m'ont donné l'opportunité de faire cette thèse dans un univers particulier et riche regroupant académique et industriel, et qui m'ont encouragé et aidé tout au long de cette expérience. Je remercie également Eric GOUBAULT, d'avoir accepté d'être président du jury; Jean-Michel BRUEL et Xavier BLANC d'être rapporteur; Jan CARLSON, Fabrice KORDON et Sébastien GÉRARD d'être membre du jury.

Je remercie aussi mes collègues de SystemX, et plus particulièrement Fateh Guenab, Frédéric Tuong, Zheng Li, Romain Gratia, Enagnon Cedric Klikpo, Aymen Boudguiga, Mohamed-Haykel Zayani, Fateh Nassim Melzi, Oussama Allali, Mouadh Yagoubi, Mostepha Khouadjia, Gauthier Fontaine, Elie Soubiran, Ali Koudri, Thomas Wouters, Laurent Wouters, Abraham Cherfi et Claude Godard avec qui l'aventure était plus intéressante. J'ai appris beaucoup en travaillant avec vous. Merci pour les discussions, les conseils, les encouragements, les matches de foot et surtout les délires qu'on a pu avoir ensemble.

Je remercie aussi énormément mes amis pour m'avoir soutenue tout au long des ces longues (très longues) années de dure labeur, et aussi pour les moments passés. Je remercie plus particulièrement : Youcef Goumiri, Younes Khadraoui, Mohamed Bouazzouni, Karim Louifi, et Rabah Guedrez.

Enfin les derniers mais pas les moindres, je remercie les membres de ma famille. je vous remercie du plus profond du cœur. Je remercie mon père Brahim pour ses conseils avisés, ses encouragements, de m'avoir poussé jusqu'au bout et sans qui rien de tout cela n'aurait pu arriver. Je te remercie infiniment. Je remercie aussi ma belle mère Ouahiba qui m'a soutenu et m'a aidé tout au long de cette thèse. Je remercie mon frère Tarik et ma soeur Nassima, mes modèles. Ils m'ont tellement apporté. Je vous remercie infiniment. Une pensée très particulière à mon exceptionnelle mère Akila. Tu as fais de moi ce que je suis aujourd'hui, et je te dédie cette thèse. Enfin je tiens a remercier mon épouse, ma moitié, Aicha qui m'a soutenue dans les moments les plus difficiles de cette thèse. Merci pour ton immense soutien dans les étapes les plus difficiles. Tu es restée à mes côtés de longues nuits de rédaction, tu m'as aidé moralement, et concrètement à finir cette thèse. Je t'aime ma femme et te remercie infiniment. Cette thèse vous appartient a vous aussi ma famille, je vous aime tous très fort, et je vous souhaite que du bonheur, et plein de succès.

Résumé

Les systèmes logiciels deviennent de plus en plus larges et complexes, et donc plus difficiles à développer et à maintenir. Cette complexité vient de la nécessité de garantir et d'améliorer un grand nombre de propriétés non fonctionnelles telles que la sûreté, la fiabilité, le temps de réponse, la maintenabilité, le coût, etc.

Afin de réduire la complexité du développement logiciel, l'ingénierie dirigée par les modèles (IDM) a proposé des techniques et des méthodes complémentaires. En IDM, les applications logicielles sont modélisées pour exprimer et évaluer des propriétés fonctionnelles et non fonctionnelles. Cependant, les propriétés non fonctionnelles sont souvent en conflit les unes avec les autres: l'amélioration d'un ensemble de propriétés non fonctionnelles peut dégrader d'autres propriétés. Par conséquent, les concepteurs logiciels doivent identifier les choix de conception appropriées et les appliquer sur des éléments architecturaux valides pour produire différentes architectures logicielles (ou alternatives architecturales). Les concepteurs doivent également vérifier si les alternatives architecturales résultantes remplissent un ensemble d'exigences et comment elles améliorent positivement les propriétés non fonctionnelles. En plus de cela, les concepteurs doivent comparer toutes les alternatives concernant leur impact sur les propriétés non fonctionnelles, et sélectionner seulement ceux qui répondent au mieux à un compromis entre les propriétés non fonctionnelles conflictuelles.

Pour créer l'espace de conception des alternatives architecturales, les concepteurs appliquent manuellement des solutions bien connues telles que les patrons de conception. Cela prend beaucoup de temps, et est sujet aux erreurs et peut-être sous-optimal. Des approches bien établies qui automatisent l'application des patrons de conception ont été introduites par l'IDM. Ces approches sont appelées transformations de modèles. Une transformation de modèle est un artefact logiciel qui spécifie un ensemble d'actions permettant de générer un modèle cible à partir d'un modèle source. Dans ce contexte, les alternatives de transformation de modèles sont des transformations de modèles qui génèrent des alternatives architecturales lorsqu'elles sont appliquées au même modèle source.

Cependant, un problème se pose lors de la création d'architectures utilisant des transformations de modèle. Plusieurs variantes de transformation de modèle doivent être identifiées afin de générer toutes les architectures logicielles possibles. Pour éviter ce problème, plusieurs travaux ont proposé de produire automatiquement de nouvelles transformations

de modèle en composant des alternatives de transformation de modèle existantes. Cette solution contribue à la production automatique d'alternatives architecturales.

Dans cette thèse, nous proposons d'explorer un espace de conception composé de compositions de transformation de modèle afin d'identifier des alternatives architecturales répondant au mieux à un compromis entre des propriétés non fonctionnelles. Cependant, l'exploration d'un espace de conception résultant de la composition des transformations du modèle peut être difficile et susceptible d'erreurs. Tout d'abord, l'espace de conception augmente rapidement avec le nombre de transformations et d'éléments architecturaux. Deuxièmement, en raison de certaines contraintes structurelles, pas toutes les compositions de transformations de modèles produisent des alternatives architecturales correctes. Troisièmement, pour des raisons de réutilisation et de maintenance, les transformations de modèles sont souvent structurées sous la forme de chaînes (c'est-à-dire que le modèle de sortie d'une transformation devient le modèle d'entrée d'une nouvelle transformation). Lorsque des alternatives de transformation de modèles existent pour chaque maillon d'une telle chaîne, plusieurs étapes d'exploration doivent être enchaînées, ce qui augmente de manière significative le problème combinatoire.

Afin de surmonter ces problèmes, nous utilisons des techniques d'optimisation multiples-objectifs pour l'exploration de l'espace de conception. Ces techniques offrent un processus permettant de comparer automatiquement des solutions par rapport à de multiples objectifs contradictoires (par exemple, des propriétés non fonctionnelles), et d'identifier un ensemble de solutions non dominées (ou solutions optimales): étant donné un ensemble d'objectifs, une solution S1 est non-dominée par une autre solution S2, si S1 est au moins aussi bonne que S2 dans tous les objectifs et, S1 est strictement meilleure que S2 au moins pour un objectif.

Dans cette thèse, nous présentons une approche qui automatise la composition des transformations de modèles à l'aide d'algorithme évolutionnistes (AEs): ces algorithmes permettent d'explorer efficacement de grands espaces de conception. Contrairement aux approches existantes, notre approche automatise l'exploration de compositions de transformation de modèles utilisées pour produire des architectures plutôt que d'explorer des architectures isolées des transformations de modèles. En conséquence, l'évaluation des architectures est réalisée sur les modèles produits par ces transformations, et l'algorithme d'optimisation (l'AE dans notre cas) peut être réutilisé pour différents types de transformations de modèle.

Mots-clés: Sûreté de fonctionnement, Ingénierie Dirigée par les Modèles, Patron de conception, Transformation de Modèle, Composition de transformations de modèles, Chaines de Transformations, Propriétés non fonctionnelles, Exploration d'espaces de conception,

techniques d'optimisation multi-objectifs, transformations de modèles à base de règles, transformations ATL, modèles AADL, NSGA-II, Solveurs SAT, programmation linéaire.

Abstract

Software systems become more and more large and complex, and therefore more difficult to develop and maintain. This complexity comes from the necessity to guarantee and improve a large number of non functional properties such as safety, reliability, timing performance, maintainability, cost, etc.

In order to reduce software development complexity, Model Driven Engineering (MDE) proposed complementary techniques and methods. In MDE, software applications are modelled in order to express and evaluate functional and non-functional properties. However, non functional properties often conflict with each other: improving a set of non functional properties requires to degrade other properties. Consequently, software designers must identify appropriate design decisions and apply them on valid architectural elements to produce different software architectures (or architecture alternatives). Designers must also check whether the resulting architecture alternatives fulfil a set of requirements and how they positively improve the non functional properties. Additionally, designers must compare all the alternatives regarding their impact on the NFPs, and select only those answering at best to a trade-off among the conflicting non functional properties.

To create the design space of architecture alternatives, designers apply well-known solutions such as design patterns manually. This is time-consuming, error-prone and possibly sub-optimal. Well-established approaches that automate the application of design patterns, were introduced by MDE. These approaches are called model transformations. A model transformation is a software artefact that specifies a set of actions to generate a target model from a source model. In this context model transformations alternatives are model transformations that generate architecture alternatives when applied to the same source model.

However a problem raises when creating architectures using model transformations. Several model transformation alternatives must be identified in order to generate all the possible software architectures. To avoid this problem, several works proposed to produce automatically new model transformations by composing existing model transformation alternatives. This solution helps in the automatic production of architecture alternatives.

In this thesis, we propose to explore a design space made up of model transformation compositions in order to identify architecture alternatives answering at best to a trade-off among conflicting NFPs. However, exploring a design space resulting from the composition of model transformations can be difficult and error-prone. Firstly, the design space grows rapidly with the number of transformations and architectural elements. Secondly,

due to some structural constraints not all the possible compositions of model transformations produce correct architecture alternatives. Thirdly, for reuse and maintainance reasons, model transformations are often structured as chains (i.e. the output model of a transformation becomes the input model of a new transformation). When transformation alternatives exist for each link in such chain, several exploration steps have to be chained, which increases significantly the combinatorial problem.

To overcome these problems, we use Multiple Objective Optimization Techniques (MOOTs) for design space exploration. These techniques offer a process to compare automatically solutions with respect to multiple conflicting objectives (e.g. non functional properties), and identify a set of non-dominated solutions (or optimal solutions): given a set of objectives, a solution $S1$ is non-dominated by another solution $S2$, if $S1$ is at least as good as $S2$ in all the objectives, and additionally, $S1$ is strictly better than $S2$ in at lest one objective.

In this thesis, we present an approach that automates the composition of model transformations using Evolutionary Algorithms (EA): such MOOTs enable to explore efficiently large design spaces. As opposed to existing approaches, our approach automates the exploration of model transformation compositions used to produce architectures rather than exploring architectures in isolation of model transformations. As a result, architectures evaluation is performed on models produced by these transformations, and the optimization algorithm (EA in our case) can be reused for different types of model transformations.

Keywords: Reliable Embedded systems, Model Driven Engineering, Design patterns, Model transformations, Model transformations composition, Model Transformation Chains, Non-functional properties, Design space exploration, Multiple-objective optimization techniques, Rule-based transformation languages, AADL models, evolutionary algorithms, NSGA-II, ATL model transformations, SAT solvers, Linear programming

Contents

List of illustrations

List of code examples

1 Introduction

Contents

1.1 Context

Predicting the non functional properties [1] of a software system before it is implemented is crucial for the success of the development process. It helps software designers to anticipate potential failures of the system, prevent expensive changes, and thus build high quality systems.

The prediction of non-functional properties (*NFPs*) of software systems has been a major research field in recent years. Several works proposed different methodologies and tools to estimate different non functional properties such as performance, reliability, cost, availability, dependability, etc. These works rely on modelling techniques. Models present several benefits: they offer a comprehensible view of a system under development. They highlight the most important artefacts (e.g., hardware components, memory, tasks) of a given system and also their interactions. In addition to all of that, models offer the possibility to analyse systems regarding different functional and non functional properties before a concrete implementation is produced. Thus, using models system designers can check whether a given system is expected to meet the required non functional properties.

Furthermore, design patterns have been proposed to improve non-functional properties of a system. Design patterns are reusable solutions to solve recurring design problems. Thus, it is useful to have knowledge of these design patterns to consider different design options, and thus analyse and improve different non functional properties.

The FSF (Fiabilité et Sûreté de Fonctionnement) [2] project is a good example of project where railway designers want to predict the NFPs of an embedded system before its implementation. This project is based on a railway usage scenario where railway designers of Alstom Transport want to organize operational software and hardware resources so that the resulting system improves the following non functional properties: safety, reliability, availability and response time.

More precisely, in this project railway designers are interested in the application of design patterns (software replication, software component allocation, model refinement) in the development of safety-critical systems. The application of these patterns will be evaluated according to their impact in terms of safety, reliability, response time, maintainability, etc. For instance, we consider the well-known design pattern triple replication of hardware components. This pattern consists in replicating a hardware component three times. It also uses a voter to (i) collect the outputs of the three replica, and (ii) determine the actual final output of the system. Thus, the voter represents a decision making artefact. The presence

[1] Specify global constraints on how the system operates or how the functionality is exhibited
[2] Safe and reliable embedded systems

of the voter increases the response time of the system, because the voter communicates with the hardware replica, and uses functions to determine the final output of the system. On the other hand, due to the use of three hardware replica the system is safer. Other patterns that replicate hardware components can also be used. Thus, different architecture alternatives are produced from the application of these patterns. These architectures are then evaluated in terms of safety, and response time in order to find the architecture that maximizes these NFPs.

In this project, we plan to create the design space of architectures using design patterns, evaluate these architectures regarding different NFPs, and compare them in order to find the optimal architecture.

1.2 Challenges

The goal of this thesis is to develop a model-based approach that helps in the prediction of the non functional properties (*NFPs*) of a software system. To develop such approach, we have to overcome the following challenges.

In many cases, NFPs conflict with each other: improving one NFP can have a negative impact on other NFPs. For instance, improving the availability of a hardware component is often done by the replication of this component, that increases the weight and also the response time of the replicated component. Similarly, allocating several interconnected software components on the same hardware component improves the response time of the system. Yet, centralize all the software components in a single hardware component decreases the reliability of the system. In these examples, improving a particular NFP decreases others NFPs. Thus, modelling a system that improves all its NFPs simultaneously is not always possible. Therefore, designers have to consider several architecture alternatives, and explore them to find the optimal architecture alternatives regarding the NFPs. In other words, designers are confronted to a **multiple objective optimization problem (MOOP)**.

A MOOP is characterized by (i) a design space of architecture alternatives, and (ii) objective functions that evaluate the quality of these alternatives. The objective functions of each alternative within the design space is computed to evaluate the quality of the alternative. As these objectives are conflicting, there is no single optimal architecture alternative but a variety of choices that represent different trade-offs. Optimality in this case is defined using the concept of Pareto-dominance: an architecture alternative dominates another one if it is equal or better in all objectives and strictly better in at least one objective. In a design space of architecture alternatives, those are called Pareto-optimal solutions, they

represent the architecture alternatives that dominate all the other alternatives. Thus, solving a MOOP consists in finding the Pareto-optimal architecture alternatives.

Evaluating each architecture alternative regarding the NFPs is complex as the design space can quickly become very large. For instance, consider an embedded system with 30 software components, and 10 hardware components. Each software component can be allocated on any hardware component, leading to a design space composed of 10^{30} architecture alternatives. The complexity increases even more when having multiple conflicting NFPs. In this case, each architecture alternative is evaluated regarding the NFPs and the results of the evaluation must be compared with the corresponding evaluation results of every other alternative. This process is called **design space exploration**.

Design space exploration is the process of exploring a search space composed of architecture alternatives to identify Pareto-optimal architecture alternatives: alternatives that satisfy at best a set of NFPs such as reliability, safety, maintainability, etc. Design space exploration is a challenging problem. In the previous years, several works proposed solutions for this problem. These solutions are based on search strategies that enumerate manually every architecture alternative in the design space, evaluate these alternatives regarding different NFPs and compare them to select the Pareto-optimal architecture alternatives. These solutions are simple to implement, but they are very costly: the more the size of the design space increases, the more difficult it becomes to enumerate all the possible architecture alternatives. Additionally, in these works, for each new multiple-objective optimization problem, the used exploration strategy has to be adapted to the new problem in order to produce the architecture alternatives and explore them. In other words, these solutions are not generic regarding different multiple-objectives optimization problems.

In this thesis, we want to provide an **automated** and **generic** exploration framework that allows us to solve different design space exploration problems involving multiple conflicting NFPs.

1.3 Problems overview

Developing a design space exploration framework that supports explorations of software architecture alternatives has the following problems.

1.3.1 Creation of the design space of architecture alternatives

As defined before, software designers reuse existing design patterns (created in previous projects) to produce architecture alternatives. Usually, these patterns are applied manually

to create architecture alternatives. To automate the application of design patterns model driven engineering introduced the concept of model transformations. Model transformations formalize in reusable artefacts the implementation of different design patterns such as design decisions, model refinements.

A model transformation specifies a set of actions to generate a target model from a source model. The goal underlying the use of model transformations is to save time and efforts and reduce errors by automating the production and modifications of models. Additionally model transformations can be reused in different projects or products without major modifications. In this context, model transformation alternatives are transformations that produce different target models when applied on the same source model. These target models represent architecture alternatives which may differ in the NFPs (e.g., reliability, response time, etc.) they exhibit. Additionally, these transformation alternatives can be composed (chained or merged) together to create new model transformations, and thus automatically produce new architecture alternatives. Model transformation is a promising solution in order to model the architecture alternatives for different design problems. It represents a generic solution that can be incorporated in an exploration framework. However, the design space of architecture alternatives grows rapidly with the number of created model transformations and elements in the model which leads to a combinatorial problem. Thus, finding the optimal architecture alternatives from all feasible alternatives becomes difficult to perform using only the composition of model transformations.

1.3.2 Correctness of architecture alternatives

When exploring a design space of architecture alternatives another problem is to check the correctness of architecture alternatives regarding different structural constraints such as memory constraints, software components co-localization, etc. Structural constraints are defined by software designers, and are necessary to guide the exploration of the design space towards correct architecture alternatives.

In the past, several works proposed solutions to check the correctness of architectures alternatives regarding a set of structural constraints. In these works, the structural constraints are checked while exploring the architecture alternatives: evaluation and comparison of architecture alternatives. More precisely, structural constraints are expressed on each architecture alternative using a high-level constraint specification language. This language is then used to check the correctness of the architecture alternative. If the alternative respect the structural constraints, then this alternative is considered in the design space to explore, and is evaluated and compared to other correct alternatives, otherwise the

alternative is rejected. This operation can take very long time to find correct architecture alternatives. In some cases this operation could not converge at all (no correct architecture alternatives). Thus, we have to provide an exploration framework that identify correct architecture alternatives while reducing the search time and converging to optimal (or near optimal) architecture alternatives.

In the previous section, we proposed to create the design space of architecture alternatives using model transformation compositions. In this case it will be more interesting to check structural constraints on model transformations before executing them, and thus produce the corresponding architecture alternatives. This solution helps to reduce the design space to only correct architecture alternatives. However, another problem raises when checking structural constraints on model transformations. Structural constraints are usually expressed on models using high-level constraint languages such Object Constraint Language (OCL). On the other hand, model transformations are represented by a code written in a programming language. Thus, we must find a solution to express structural constraints on model transformation compositions in order to produce correct architecture alternatives.

1.4 Contributions

The objective of this thesis is to build a framework for design space exploration that can be used to solve different multiple-objective optimization problems. The proposed framework is based on two research fields: model transformations and multiple-objective optimization techniques.

To automate the production of the architecture alternatives, we propose to use model transformations. More precisely, we propose to compose model transformation alternatives in order to automate the production and generation of architecture alternatives. However, not all compositions of model transformations generate correct architecture alternatives. To solve this problem, we validate a-priori the composition of model transformations: we check the structural constraints satisfiability at the construction of model transformation composites. To perform that, we formalize structural constraints on model transformations using boolean formula, and apply a SAT solving technique to extract valid model transformation alternatives. Performing this a-priori validation, we ensure that the resulting architecture alternatives are correct, and thus reduce the design space of architecture alternatives.

In order to solve the multiple-objective optimization problem and thus find Pareto-optimal architecture alternative, we have to explore the created design space made up of

model transformations. To perform that, we propose to use multiple-objectives optimization techniques. More precisely, we encode an optimization problem composed of model transformation compositions in a specific multiple-objectives optimization technique: evolutionary algorithms (EAs). The proposed encoding considers the a-priori validation of model transformations in order to generate correct architecture alternatives. Applying the evolutionary algorithm, we find composite transformations generating optimal (or near optimal) architecture alternatives.

For reuse and maintainability reasons we chain model transformations together by providing the output (i.e. the intermediate model) of one transformation as input of another transformation. Thus, to find a set of optimal architecture alternatives, Pareto-optimal chained model transformation alternatives must be identified and executed. To identify these alternatives, we rely on evolutionary algorithms.

Finally, to evaluate the quality and genericity of the proposed approach, we lay different evaluation criteria, and apply our framework on a set of experimental case studies (including railway case studies). Using the experimental results we answer to the predefined evaluation criteria, and thus evaluate our framework.

1.5 Structure of this Thesis

This thesis is structured as follows:

In chapter 2, we discuss related works. In the first section of chapter 2 we discuss related approaches that target the formalization of architecture alternatives using model transformations. Then, the second section of chapter 2 describes the different multi-objective optimization techniques used to solve a given optimization problem. In the third section, we present and study different works that apply multiple-objective optimization techniques on a design space of architecture alternatives. Finally, in the fourth section we discuss related approaches for the use of model transformations during design space exploration.

In chapter 3, we detail the problems concerning the optimization of software architectures regarding a set of conflicting non functional properties, and show why existing solutions are not satisfactory.

In chapter 4, we provide an overview of our proposals to resolve the identified problems. Firstly, we demonstrate how we identify model transformation alternatives, and

compose them to automate the identification and production of architecture alternatives. Secondly, we present our solution to explore a design space of composite model transformations and identify optimal (or near optimal) architecture alternatives. Finally, we define how we automate the application of several design patterns, and explore the resulting design space.

In chapter 5, we explain our solution to automate the identification and production of architecture alternatives. The proposed solution consists in applying rule-based model transformations, and compose these model transformations while considering a set of structural constraints.

In chapter 6, we describe our exploration engine dedicated to find optimal (or near optimal) architecture alternatives. This engine is based on evolutionary algorithms and model transformation composition techniques. In this chapter we present the principles of evolutionary algorithms. Then, we describe how evolutionary algorithms are applied to an optimization problem involving the application of a unique rule based model transformation. Finally, we extended our approach by applying evolutionary algorithms on an optimization problem involving the chaining of several rule based model transformations.

In chapter 7, we describe the evaluation and validation of our approach. To evaluate and validate our approach, we first established a set of evaluation criteria. These criteria are used to evaluate (i) the genericity of our approach, (ii) the performance of the exploration engine quantitatively, and (iii) the resource consumption of the engine. To validate these criteria, we applied our approach on railway case studies having different multiple objective optimization problems.

In chapter 8, we conclude this thesis by summarizing the contributions of this thesis, and present possible future works.

2 Related Works

Contents

The objective of this thesis is to create a generic and automated framework for design space exploration that can be used to solve different multiple-objective optimization problems. In the past, several works proposed approaches and frameworks to solve different design space exploration problems. In this chapter we give an overview of these approaches. The work presented in this thesis covers two main fields: model transformations and multiple-objective optimization techniques. In section 2.1, we focus on model transformation approaches in general. These approaches allow to automate the production of software architectures. In section 2.2, we describe the basics of multi-objective optimization techniques. In section 2.3, we present background material on frameworks that tackle architectural optimization problems using multi-objective optimization techniques. Finally, in section 2.4 we present related works aiming at automating design space exploration using model transformations.

2.1 Formalization and analysis of software architecture alternatives

An important goal of this thesis is to help system architects to automate the production of software architecture alternatives, and evaluate these alternatives regarding a set of conflicting non functional properties. In the past, different solutions have been proposed to perform the formalization and analysis of architecture alternatives. Some works such as in [1] proposed an approach based on feature models to capture configurations (or alternatives) in software produce lines [2]. Feature models allow to manage variable and common properties of products in a software product line, and to derive alternatives of software systems. Other works such as in [3] proposed an approach based on model transformations to automate the production of architecture alternatives for single-software systems. Model transformations allow to produce new models (architecture alternatives) from existing ones. In the following we detail these two different techniques: feature models and model transformations.

2.1.1 Feature models

Feature models are approaches for representing commonalities, variation, and configuration of software product lines. They are based on features (or characteristics such as requirements, non-functional properties), to identify all possible product configurations of software product lines. More precisely, feature models have a tree structure, where nodes represent the features, and the edges represent the dependencies (structural constraints)

among the features. Given three features f_1, f_2 and f_3. Assume the features f_1 and f_2 are interdependent. If the feature f_1 is selected, then the feature f_2 must be selected and the feature f_3 excluded. Performing valid combinations of features lead to different product configurations. Each product configuration is then measured to assess its non-functional properties.

In the past, some works [4; 1] proposed methods based on feature models to evaluate the non-functional properties of software product lines (SPLs). In these works, the authors stated that feature modelling are promising solutions to capture variability in software product lines and single software systems. Yet, using feature modelling the identified product configurations or architecture alternatives lack architectural details in order to compute efficiently the non functional properties. Another category of works propose the use of model transformations to raise the level of abstraction of architecture alternatives, and thus facilitate the computation of non functional properties.

2.1.2 Model transformations

In [5] the authors define model transformations as components that take models as source and produce other models as target. The source and target models can be expressed either in the same modelling languages (endogenous) or in different languages (exogenous). Thus, applying model transformations, software designers can create automatically the design space of all the possibilities (architecture alternatives). In addition to that, being reusable software artefacts the defined model transformations can be applied in different projects.

Generally, a model transformation is represented by the pattern depicted in Figure 2.1. To perform a model transformation, system designers need both the source model and a **transformation function** (i.e., a set of instructions that determine how source model elements are actually translated into target model elements).

Many works have been done in this area and several model transformation languages have been proposed to specify transformation functions. The OMG has even released the QVT (Query/View/Transformation) standard transformation language. Other languages were also proposed such as Kermeta [6], GreaT [7], ATL [8], etc.

All these languages differ concerning the way to express transformation functions. These languages are usually classified in three categories: imperative, declarative or hybrid languages.

1. *Imperative languages*: these languages [5] propose a syntax and an execution semantic similar to programming languages (Java, C++, Ada). Transformations for-

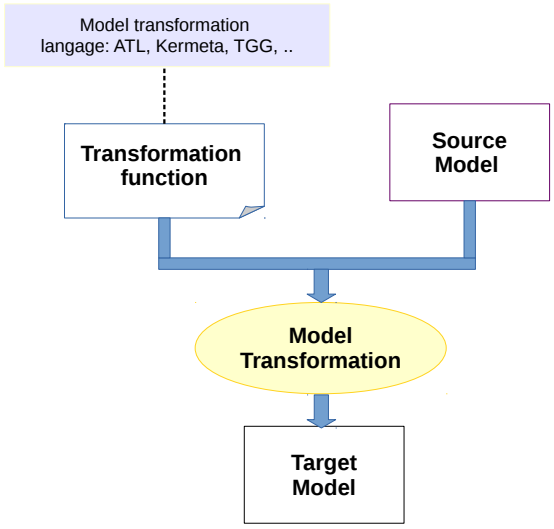

Figure 2.1: Overview of a model transformation

malized with such languages define the explicit sequence of actions to be executed in order to perform the transformation, and the languages usually provide all the standard constructs available for that purpose (loops, conditional branches, etc.). In addition, this type of model transformation languages offer a well defined control flow [9]. In other words, all the instructions in the transformation code are executed in a pre-determined order. Because of the similarity to programming languages, imperative languages are easier to learn, and can be used easily to specify complex transformations. However, to specify large and complex transformations, a large transformation code has to be written. Thus, the transformation code can quickly become difficult to read and maintain. An example of such languages are Kermeta [6], SmartQVT [10].

2. *Declarative (or Relational) languages*: these languages [5] define the relationships that must be satisfied by the elements in the source and in the target models of the transformation. It is then the responsibility of an *execution engine* to translate those relationships into an executable sequence of actions and to perform them to produce the target models. This type of model transformation languages do not offer a control flow [9]. There is a partial order among the instructions of a transformation. Consequently, the transformation code is more compact, and more comprehensible. This characteristic helps to easily specify simple transformations. An example of such languages are: Triple Graph Grammars(or TGGs) [11], .

3. *Hybrid languages*: both declarative and imperative languages have their advantages and drawbacks. Imperative languages are easy to use and expressive, but very verbose (i.e., too much code to specify simple transformations). Declarative languages on the other hand provide a code easy to understand and more compact, but less flexible and thus requiring more efforts to learn and master. For this reason, some languages called hybrid languages merged characteristics of both transformation languages. The main advantage of hybrid languages is to use the advantages of the different transformations languages, and thus compensate the weaknesses. More precisely, each hybrid language use the characteristics of one main transformation language (e.g declarative), and only use the characteristics of the other transformation language (e.g imperative) to address situations in which the main language is not suitable. An example of such languages are: the ATLAS Transformation Language (ATL) [8], the Epsilon Transformation Language (ETL) [12], UML-RSDS [13].

The main advantage of hybrid languages resides in their wide applicability range. They take into account the strengths of the imperative and declarative languages in order to facilitate the specification of transformation functions. Within hybrid languages the rule-based model transformation languages, such as ATL [8], represent very promising transformation languages. Rule-based model transformations use a clear syntax which contains both declarative and imperative elements. This gives to the transformation developer the capability to use the imperative language for complex parts of a given transformation which cannot be easily expressed using the declarative language. For these reasons, we choose to focus on hybrid languages. More precisely, we consider the hybrid language ATL, because it is mainly used in the framework RAMSES (Refinement of AADL Models for Synthesis of Embedded System) [14], a component-framework for critical embedded systems. This framework will help us to evaluate/assess the contributions of this thesis. Additionally, the semantics of hybrid languages is simpler. Thus, we can compose more easily model transformations.

2.1.3 Model transformations composition

The composition of rule-based model transformations presents several benefits. Firstly, using this technique large and complex model transformation functions can be structured as a set of small and maintainable transformations. Secondly, small transformation functions can be reused for different design problems. Thirdly, new rule-based model transformations can be created by composing/merging together existing model transformations [15; 16]. Thanks to the third benefit, software designers can apply a set of rule-based model

transformations on a source model, create new model transformations by merging the existing ones.

Composition of model transformations is a difficult problem, intensely studied in the last decade. Two types of composition techniques were already proposed for model transformations [17]: internal and external composition. In [18], the authors describe the internal composition of model transformations as the ability to compose transformation rules written in the same transformation language. It consists in location of transformation rule conflicts (e.g., rules with the same name and same context), and the definition of the execution order of these rules with the purpose to merge transformation rules into a unique transformation rule. External composition offers a different level of abstraction than the internal composition. It consists of chaining rule-based model transformations written in the same or different transformation languages by passing models from one transformation to another.

2.1.3.1 Internal composition

Internal composition techniques [17] compose two or more model transformations into one new model transformation with a merge of the transformation rules. In [17], the authors propose an internal composition technique called *Higher Order Transformations (HOTs)*. This technique is based on the superimposition technique [19] to merge two or more model transformations into a single output transformation. Superimposition is a simple internal composition in which a model transformation M_1 is put on top of another transformation M_2 to obtain a new model transformation M_3, such that:

1. M_3 contains the union of the sets of transformation rules of M_1 and M_2

2. M_3 does not contain any transformation rule of M_2 for which M_1 contains a transformation rule with the same name and the same context.

Internal composition of rule-based model transformations is an interesting solution for creating new executable model transformations, and thus automating the production of architecture alternatives. However, the internal composition of model transformations can quickly become difficult to perform, and less maintainable with the increase of the transformation rules to merge. In this context, other works [15; 16] propose the use of model transformation chains.

2.1.3.2 External composition (or model transformation chains)

External composition [15] deals with chaining separate model transformations together: the target of a model transformation becomes the source model of the next transformation of the chain. More precisely, chaining three model transformations works as follows. A model transformation is first applied on the source model to generate a corresponding intermediate model. Then a second model transformation is applied on the intermediate model and generate a second intermediate model. Finally, the target model is generated by applying the third model transformation on the second intermediate model.

However model transformations cannot be chained anyhow. In some cases, a valid order of model transformations must be established. Indeed, when chaining two model transformations T1 and T2, the target model obtained when executing the chain T1-T2 is not necessarily the same when executing the chain T2-T1. Thus, not all model transformations are commutative (i.e., can be applied in any order and produce the same target models), and dependencies (or interactions) among model transformations must be captured.

2.1.4 Discussion

In this section, we studied different modelling techniques for the formalization and analysis of architecture alternatives. We first studied feature models. This modelling technique is mostly used to generate software product configurations (alternatives) in software product lines. This approach is very promising to identify and extract different product configurations. However, it lacks accuracy when analysing the non functional properties of the product configurations.

Then, we studied model transformations. This technique present several benefits. Firstly, it allows to create new models from existing models. Secondly, model transformations can be reused in different design problems, and thus offer a generic solution. Finally, model transformations propose different composition techniques (internal and external) that allow to produce new model transformations by merging or chaining exiting ones, and thus automate the identification and generation of architecture alternatives.

When exploring the design space of architecture alternatives, software designers have to check the correctness of each generated architecture alternative. With large design spaces, this operation can be laborious. Composing model transformations allow to create the design space of architecture alternatives. However, the design space of architecture alternatives grows rapidly with the number of created model transformations and elements in

the model which leads to a combinatorial problem. Thus, finding the most suitable architecture alternatives from all feasible alternatives become difficult and sometimes impossible. Additionally, architecture alternatives are evaluated according to multiple conflicting NFPs. Thus, there is no single optimal architecture alternative but a variety of choices that represent different trade-offs. To solve such multiple-objective optimization problems, a solution is to use multiple objective optimization techniques. These techniques are described in the next subsection.

2.2 Multiple objective optimization techniques

Multiple objective optimization techniques (MOOTs) are well-known search strategies which offer a methodology and guidelines to resolve different multi-objective optimization problems. More precisely, MOOTs are used to explore large search spaces, compare the solutions of the search spaces with respect to multiple objectives (or criteria); and collect optimal solutions.

Multi-objective optimization techniques are a well established research field [20]. Most research works in this field focused on specific optimization problems, aiming at improving multiple criteria. Different MOOTs are presented in the following subsections.

2.2.1 Linear programming with weighted objectives

Linear programming with weighted objectives (LPWO) [21] is an optimization technique for finding an optimal solution to different multiple-objective optimization problems such as organization and allocation of resources. To apply LPWO on a given optimization problem, the latter must be represented by linear functions ($y = f(x)$). Thus, all the data of a multiple-objectives optimization problem (solutions, objectives and constraints) must be linearised (i.e., formulate using linear functions) into a single objective function, in order to solve the problem using LPWO. Notice that to linearise the objectives of an optimization problem, the user performs a well-known aggregation technique called weighted sum [21] based on weights and summation to have a single objective function. Applying the linear programming algorithm on this single objective function allows to find an optimal solution.

In the past, several works [22; 23; 24], proposed the use of linear programming with weighted objectives to solve different multiple objective optimization problems: resource allocation for very large scale service centers, optimization of web service compositions, In these works, the authors formalized their optimization problems as a multi-objective

0-1 integer linear problem [25], and apply linear programming algorithms to solve their problems.

Linear programming with weighted objectives provides an appropriate and simple representation (based on linear functions) of multiple-objective optimization problems. Yet, there are some limitations in the use of LPWO for multiple-objective optimization problems. Firstly, for each optimization problem, an additional effort is required to linearise the problem. Not all optimization problems can be formalized using linear functions. Some problems are composed of non-linear objectives. For instance, the objective function used to compute the reliability of an embedded system is non-linear, because the reliability of hardware components are variables, and multiplying them creates a curvilinear relationship. Applying LPWO on such problems is not efficient, and provides only one solution which is not optimal. Secondly, the application of LPWO requires to put weights on all the objectives and aggregate them into a single objective function using weighted sum. Note that weights are set manually. Thus, the identification of an optimal solution requires to put different weights for each objective and perform several experiments.

To solve multiple-objective optimization problems (including non-linear problems) other MOOTs were proposed such as evolutionary algorithms.

2.2.2 Evolutionary algorithms

Evolutionary algorithms (EAs) provide a way to search through a large number of possibilities. EAs are inspired from the natural evolution. In other words, they are based on populations that evolve over several generations. In each generation, the EA selects from the search space several interesting solutions (also called parent solutions). These solutions are then analyzed regarding different objectives (criteria) to determine their quality. From the evaluated parents, new solutions (also called offspring solutions) are determined, using selection (comparison of evaluated solutions) and genetic operators (crossover and mutation). The offspring solutions are evaluated to determine their quality and added to the population. Then, the population composed of parent and offspring solutions is shrunk by eliminating the worst solutions (regarding the results of the evaluation). Finally, all these steps (parent creation, evaluation) are repeated until the algorithm converge. Usually, an EA converges when there is no improvement in the objectives analysis of the population from one generation to another.

EAs help to study different combinations of solutions by applying genetic operators (mutation and crossover), and also consider non-linear problems. In addition to all of that, EAs were applied to a large variety of searching and optimization problems in various

fields: chemical kinetics, design of water resource systems, organization of embedded system resources. Finally, several works [26; 27] provide a good comparison of evolutionary algorithms with other multiple-objective optimization techniques. In these works, authors showed that EAs are useful in solving different multiple-objective optimization problems. They also stated that EAs are more efficient and reliable than other optimization techniques when solving various multiple-objective optimization problems related to water resources: reservoir optimization, problems of optimal allocation of resources and planning, calibration of models. However, the difficulty of using EAs resides in the application of its genetic operators. A specific formulation of design solutions is required in order to apply EAs genetic operators on them.

2.2.3 Discussion

In this section, we studied different multiple-objective optimization techniques. We first studied linear programming with weighted objectives. This optimization technique is known to be very efficient. It proposes a simple formulation based on linear functions to represent the objective functions of a given multiple-objective optimization problem. However, in this thesis we consider non-linear objective functions (e.g., reliability, response time), and linear programming with weighted objectives is not usable to solve problems having non-linear data: we have to simplify the optimization problem by putting different weights on all the objectives functions which lead to not optimal solutions.

Then, we studied another multiple-objective optimization technique: evolutionary algorithms. These algorithms are very promising to solve multiple-objective optimization problems composed of non-linear objective functions. They are based on selection, recombination and mutation mechanisms that evolve design solutions (model transformations) until producing optimal (or near optimal) solutions. However, the problem when using EAs is the genetic operators. They cannot be applied directly on the design solutions without formulating a specific encoding for the solutions. Thus, in order to apply EAs on a design space of model transformations we need a specific encoding for the transformations.

In the previous sections we described solutions to create a design space of architecture alternative using modelling techniques, and solutions to optimize multiple-objective optimization problems: MOOTs. These two different kind of solutions can be merged in order to facilitate the resolution of multiple-objective optimization problems. In the following section we describe works that were interested in merging these two different fields.

2.3 Exploring a design space of architecture alternatives

In the past, several research works were interested in exploring a design space composed of architecture alternatives using multiple-objective optimization techniques. These works provide different exploration frameworks that model (using UML, UML-RT, AADL, PCM, LQN) different software architecture alternatives, analyse these alternatives regarding a set of conflicting non functional properties, explore the design space of architecture alternatives, and extract optimal (or near optimal) architecture alternatives. In these works, the authors targeted different concerns:

1. In [28] the authors took into account structural constraints (defined by software designers) in order to check the correctness of the architecture alternatives, and thus reduce the size of the search space to explore.

2. In some works [29; 30], the authors proposed generic exploration frameworks. The authors stated that their frameworks can be re-used without major modifications to solve different multiple-objective optimization problems.

Using these concerns we categorized the studied frameworks. In the following subsections we describe these works.

2.3.1 Managing structural constraints

Aleti et al. [31; 28; 32] developed a framework called ArcheOpterix which optimizes architecture models by means of multi-objective optimisation techniques such as evolutionary algorithms, ant colonies, or simulated annealing. It works on the basis of AADL (Architecture Analysis & Design Language) specifications [33]. The framework ArcheOpterix has been mainly used on optimization problems composed of three non functional properties (data transmission reliability [32], communication overhead [28], and energy [28]). Additionally, ArcheOpterix is based on analytical methods (e.g., Discrete Time Markov Chain) to evaluate the properties, optimisation and constraints validation (e.g., memory and co-localization constraints). The ArcheOpterix framework is not generic according to the optimization problems: for each new optimization problem the new architecture alternatives must be mapped into the used evolutionary algorithm in order to solve the optimization problem. Thus, ArcheOpterix is limited to the studied problems: software allocation problems.

Fredriksson et al. [34] developed a framework which optimizes the allocation of components to real time tasks by means of of multi-objective optimisation techniques such

as evolutionary algorithms, bin packing [35] and simulated annealing. This framework has been used during compile time to minimize resource usage and maximize timeliness. When creating the design space of architecture alternatives, the authors considered different structural constraints such as software component isolation, intersecting transactions. However, this framework is not generic in accordance with the optimization problems: this framework was developed to solve only one type of optimization problem: tasks allocation. In order to solve different problems this framework has to be re-written.

2.3.2 Genericity of design space exploration frameworks

Martens et al. [36; 37; 29] developed a framework called PerOpteryx, using the Palladio Component Model (PCM) [38] to describe software architectures. This framework proposes to use evolutionary algorithms, combining genetic operators and so called tactics [37], in order to optimize architectures regarding a set of non functional properties: performance, reliability and cost. In the PerOpteryx framework, tactics encode the expertise of designers with well identified design patterns. Tactics in PerOpteryx are similar to model transformations. However, tactics are internal operators of PerOpteryx and their applicability is detected dynamically during the optimization process. As a consequence, it is difficult to constrain the composition of tactics. Additionally, in PerOpteryx all the possible architecture alternatives (correct and incorrect regarding structural constraints) are considered in the design space. When applying the evolutionary algorithm the correctness of architecture alternatives is checked dynamically and the incorrect alternatives are rejected. This checking process can be very time-consuming when having a large number of incorrect architecture alternatives. In addition, PerOpteryx is not really generic according to the modelling languages: the framework is only applied for software systems modelled using PCM.

Li et al. [30] developed a framework called AQOSA (Automated Quality driven Optimization of Software Architecture) for the optimization of architecture models by means of evolutionary algorithms. This framework is based on AADL [33] to describe software architectures. It supports multiple degrees of freedom (architecture topology, Allocation of service instances, hardware components replacement) for automatically generating architecture alternatives. It also integrates different performance analysis methods to improve two conflicting non functional properties: data transmission reliability and communication overhead. To demonstrate the applicability of their framework, the authors performed experiments on a small scale and a large scale real world case studies: the car radio navigation (CRN) system and the business reporting system (BRS). From the obtained results,

the authors stated that AQOSA (i) helps software designers to reduce the workload for modeling; and (ii) improves non functional properties of designed models by using evolutionary algorithms. In addition to the experiment part, the authors also provide a comparison of results obtained with different versions of EAs (NSGA-II [39], SPEA-II [40], and SMS-EMOA [41]). However, in this work the authors assume that architecture models are correct, and thus they do not check the validity of these models regarding a set of structural constraints. Structural constraints allow to put more details in architecture models by restricting design decisions. Therefore, AQOSA cannot be applied for optimization problems where software designers proposed a set of structural constraints.

2.3.3 Discussion

In this section, we described and studied different exploration frameworks that combine multiple-objective optimization techniques and software architectures prediction. We observed that these works vary in scope and concerns.

The first category of works [28; 34] provide methods to create a design space of architecture alternatives while considering different structural constraints. However, these works lack genericity in order to solve different multiple-objective optimization problems. In other words, these works are limited to one multiple-objective optimization problem. In order to solve other optimization problems, the authors have to re-write their exploration frameworks.

The second category of works [29; 30] propose generic exploration frameworks. These frameworks can be used to solve different optimization problems without performing great changes. However, in [29] the authors considered all the possible (correct and incorrect) architecture alternatives in the design space, and when exploring rejected the incorrect alternatives. This operation is time-consuming especially when the design space is composed of several incorrect architecture alternatives. Additionally, to use the framework presented in [29], the users have to learn or master the PCM modelling language. In [30], the authors took assumption that the architecture alternatives are correct without taking into account possible structural constraints.

As a result, we observe that none of the studied exploration frameworks overcome all the identified challenges of this thesis. Open limitations are (i) the consideration of structural constraints, and (ii) the genericity of the exploration frameworks regarding optimization problems and modelling languages.

In the described exploration frameworks, the authors used modelling languages to produce the search space of architecture alternatives, and optimization algorithms to identify

optimal alternatives. Another category of works proposed the use of model driven engineering techniques such as model transformations to automate the production of architecture alternatives. More precisely, they proposed to explore a design space of model transformations in order to identify optimal (or near optimal) architecture alternatives. In the following subsection we present and study these exploration frameworks.

2.4 Exploring a design space of model transformations

In the past, few research works were interested in using model transformations techniques for improvement of software architectures. These works introduce different exploration engines that (i) explore a design space of model transformations, where each transformation generate a different architecture alternative, (ii) analyse model transformations regarding a set of NFPs, and (iii) apply a search strategy to identify the model transformations that generate optimal (or near optimal) architecture alternative. All these works proposed interesting approaches for design space exploration and targeted different concerns:

1. Model transformations formalize in reusable artefacts the implementation of different design patterns: safety patterns, allocation of software components, model refinement. In the past, several works proposed to implement one design pattern at a time. For instance, they cannot produce a system implementing safety patterns and design decisions together. In the studied exploration frameworks [42; 43; 44], the authors proposed an approach to consider several design patterns, and thus study more complex software architectures.

2. In many cases model transformations are interdependent, and their composition should take complex dependency relation into account in order to generate correct software architecture alternatives. In some works [45; 46; 47], the authors proposed solutions to consider model transformations dependencies when producing architecture alternatives.

3. Some works [48; 49; 50; 51] were interested in the exploration of a design space of model transformations. They proposed approaches to map model transformations into different exploration strategies or multiple-objective optimization techniques, and identify the model transformations generating the optimal (or near optimal) architecture alternatives.

Using these three concerns we categorized the studied approaches. In the following subsections we describe these works.

2.4.1 Automate several steps of a design process

Insfran et al. [42] proposed a semi-automatic approach based on model transformations to produce and select architectural alternatives. This approach requires to know a priori the impact of model transformations on non functional properties of produced architectures. With complex architectures and model transformations, this knowledge might be difficult to formalize. Similarly, Loniewski et al. [52] proposed a semi-automatic approach, based on an internal composition of model transformations to automate the production of architecture alternatives that answer to a trade off among conflicting non functional properties. However, this approach is limited to internal model transformations. In addition to that, the authors do not take into account dependencies among model transformations when composing them. Finally, none of the mentioned approaches in [42] and [52] is fully automated. This becomes a strong limitation when considering huge design spaces.

Li et al. [53; 43] developed a framework called PETUT-MOO (Performance-Enhancing Tool using UML Transformations and Multi-objective Optimizations) that provides modelling (UML), model transformation compositions, optimization and evaluation techniques to optimize software architectures regarding non functional properties. The authors emphasizes that PETUT-MOO is generic regarding different modelling techniques: any evaluation technique for a given non functional property based on UML can be plugged in the framework. In addition to that, model transformations that implement different design patterns can be integrated in the framework. However, in this work assume that the composition of model transformations generate correct architecture alternatives. Model transformations can interact with each other, and their composition may lead to invalid new model transformations: that generate incorrect architecture alternatives.

Mirandola et al. [44] proposed an approach based on model transformation compositions to deal with non functional properties of software architectures. Given a set of architectural patterns, and a catalogue of model transformations, a method was proposed to guide architects towards architectures that comply with non functional properties. However, this method is limited to internal composition of model transformations (see the description in subsection 2.1.3.1).

2.4.2 Management of model transformation dependencies

Schatz et al. [45] developed an approach which optimizes the allocation of software architectures to hardware platforms by means of declarative transformation rules, and structural constraints management. More precisely, the application of each transformation rule generates a potential architecture alternative. The correctness of this alternative is then

checked regarding a set of structural constraints. However, this approach does not use any optimization algorithm when exploring the design space. Transformations rules and constraints management guide the user to optimal architecture alternatives. Thus, this approach is not automated, which is a strong limitation when considering huge design spaces.

Ciancone et al. [46] developed a framework called Klapersuite that offers system designers a means to analyse two non functional properties (performance and reliability) at early stages of the development process. The approach uses model transformations to automatically generate analysis models (using queuing networks) out of an annotated UML design model. The approach focuses on validation of non functional properties during early stages of a design process and does not offer any optimization technique to explore and select the best analysis models.

Drago et al. [47] proposed an extension of the model transformation language *qvt-relations* in order to automate design space exploration from the description of model transformations. However, to the best of our knowledge, the exploration engine proposed in this work does not rely on any well identified optimization technique. The authors use a semi-automatic exploration strategy, which becomes a strong limitation when considering huge design spaces.

2.4.3 Automatic exploration of model transformations

Kavimandan et al [48] presented an approach to optimise real-time QoS configuration in distributed and embedded systems. This approach is based on model transformations to formalize architecture alternatives, and bin packing algorithms to explore the design space and identify optimal software architectures. In this work, the authors developed a specific framework that consider only one design pattern: component allocation. To consider another design pattern, the authors have to re-write their framework. Thus, the proposed approach is not generic regarding different design patterns.

Denil et al. [50] proposed an approach that optimizes architecture models by means of rule-based model transformations and multiple objective optimization techniques such as simulated annealing, and hill climbing. In this approach architecture alternatives are generated by executing one model transformation or by performing an internal composition of model transformations. Generated architecture alternatives are then checked regarding different structural constraints in order to reduce the design space to explore. Simulated annealing or hill climbing is then applied on the design space of feasible architecture alternatives to identify the optimal alternatives. In this approach the authors do not take into

account the possible dependencies among model transformations before generating architecture alternatives. Instead of that, they systematically execute each transformation, and check the correctness of the produced architecture alternatives. This operation is time-consuming. In some cases, the optimization algorithm can take very long time to find correct architecture alternatives or could not converge at all (no feasible solutions).

In [49; 51], the authors presented exploration frameworks that improves the performance of software architectures using exploration strategies techniques and rule-based approaches. More precisely, the authors propose the use of transformation rules to formalize different software architecture alternatives, discover interactions (dependencies) among software components and thus provide improvements for software architectures. In these works, the authors provide their own search algorithm for design space exploration. Yet, the efficiency of the provided search algorithm has not been proven nor the algorithm has been compared to other widely used search algorithms.

2.4.4 Discussion

In this section, we described and studied different exploration frameworks that improve the production of software architectures using model transformation techniques and exploration search strategies. We observed that these works vary in scope and concerns.

The first category of works provide methods based on model transformation compositions to automate several steps of a design process. However, these methods present certain limitations. In these works the authors do not take into account possible structural constraints among model transformations when composing them. Consequently, they consider incorrect architecture alternatives when exploring the design space. Additionally, these works consider only internal composition of model transformations.

The second category of works provide methods to compose model transformations while taking into account different structural constraints. However, these works do not propose a fully automatic approach to explore the design space of model transformations. They use semi-automatic exploration strategies to guide the designers to optimal (or near optimal) architecture alternatives. These exploration strategies present a strong limitation when exploring large design spaces.

The third category of works provide frameworks that combine multiple-objective optimization techniques and model transformations techniques to solve a design space exploration problem. However, these frameworks present different limitations. In [48], the exploration framework lacks genericity in order to solve different multiple-objective optimization problems. In other words, these works are limited to the application of one

design pattern: component allocation. In [50], the authors do not take into account structural constraints among model transformations when composing them. They create new model transformations execute them and check the correctness of the resulting architecture alternatives. The exploration can take very long time to find correct architecture alternatives, or could not converge at all (no correct alternative has been found). In [49; 51], the authors use their own exploration strategies to explore the design space of model transformations. These exploration strategies have not been proven nor compared to other widely used multiple-objective optimization algorithms.

None of the studied works overcome all the identified challenges of this thesis. Open limitations are (i) the a-priori validation of model transformation compositions regarding different structural constraints, and (ii) the automatic exploration of a design space of model transformations.

2.5 Conclusion

Solving a design space exploration problem consists in finding correct and optimal (or near optimal) solutions given a design space of feasible solutions, conflicting non-functional properties and a set of structural constraints. In the past, several works proposed different methodologies or frameworks to solve some or all parts of a design space exploration problem.

Some works proposed to automate the creation of the design space using different modelling techniques: feature models and model transformations. Feature models are very promising to capture the variability of software product lines and single software systems. Yet, they are not very accurate when computing the non functional properties of a given system. Model transformations are solutions that produce models from other models. Additionally, they propose compositions techniques that allow to create dynamically new model transformations from existing ones and thus automate the production of architecture alternatives. This technique is very promising to create a design space of architecture alternatives. However, using only model transformations, software designers cannot explore efficiently a design space: they have to execute each transformation, compare together the resulting architecture alternatives and identify optimal (or near optimal) alternatives. To solve the latter issue several works proposed to combine modelling techniques and multiple-objective optimization techniques.

A first category of works [29; 30; 28; 34] proposed to combine modelling languages such as AADL, UML and PCM with multiple objective optimization techniques to solve different optimization problems. In these works, the authors targeted different concerns:

(i) the management of structural constraints proposed by the designers, and (ii) the development of a generic framework to solve different optimization problems without major modifications. However, the proposed exploration frameworks are not fully generic regarding the modelling techniques and the optimization problems. Additionally, they propose to check the structural constraints while exploring the architecture alternatives. This operation can take very long time to find correct architecture alternatives or could not converge at all (no feasible solutions).

Another category of works [52; 43; 44; 47; 50] proposed to use model transformation compositions to solve different multiple-objective optimization problems. In these works, the authors targeted different concerns: (i) the implementation of several design patterns; (ii) the composition of model transformations while considering possible dependencies and (iii) the exploration of a design space of model transformations using different exploration strategies. However, the proposed approaches do not take into account the dependencies among model transformations when composing them. Consequently, the authors consider incorrect architecture alternatives when exploring the design space. With large design spaces the optimization algorithm may not converge at all (no feasible solutions). Additionally, some works are based on semi-automatic methods to explore the design space of model transformations. These works are very time consuming and error prone.

In this thesis, we consider the advantages of all the presented approaches while tackling their limitations in order to solve different multiple-objective optimization problems. More precisely, we suggest the use of model transformation compositions to automate the production of architecture alternatives. This modelling technique can be plugged into an evolutionary algorithm and thus be combined with genetic operators such as mutation and crossover. To the best of our knowledge, combing model transformation compositions and evolutionary algorithms have not been described before.

However, there are some challenges that we must surpass in order to attain these objectives. Firstly, composing two or more model transformations together do not necessarily produce a valid model transformation, i.e., which generates an architecture alternative that respect some structural constraints. Thus, when composing model transformations we need to check the validity of the produced model transformations before generating the corresponding architecture alternative. Secondly, model transformations are represented by a code written in a transformation language (see subsection 2.1.2). On the other hand, MOOTs are applied on solutions represented by specific encoding (binary, real numbers). Thus, we must find an encoding for the composition of model transformations in order to apply MOOTs on them.

In the following chapter, we first detail these challenges. Then, we provide and criticize possible solutions for each challenge. Finally, we define what are the solutions we have adopted to overcome these challenges.

3 Problem Statement

CONTENTS

3.1 Introduction

Software systems become more and more large and complex, and therefore more difficult to develop and maintain. Part of this complexity comes from the necessity to guarantee and improve a large number of **Non Functional Properties (NFPs)** such as the safety, reliability, timing performance, energy consumption, etc.

To overcome this complexity, **Model Driven Engineering (MDE)** proposes the use of **models** as a key asset in the development of complex systems. In MDE, models are used to specify, simulate, verify, analyze, test and generate the system to be built. More precisely, models offer a higher abstraction representation of the system, which eases the prediction of NFPs before a concrete implementation is available. In this context, **model transformations** play a very important role: they implement in an automated manner reusable manipulations of a model that implement recurrent design decisions.

Design decisions impact significantly the NFPs of the system. Most of these decisions have to be taken in different products, finding different **architecture alternatives** depending on the context in which they are considered. In addition to that NFPs of a system often compete with each other: improving one NFP requires to degrade other NFPs. As a consequence, a trade-off has to be met among these NFPs, and **model transformation alternatives** have to be identified, evaluated and selected.

However a problem raises when formalizing architecture alternatives using model transformations: we must identify several transformation alternatives in order to generate all the possible architecture alternatives. To avoid this problem, some works [19; 54] proposed to produce automatically new model transformations (called composite transformations) by **composing** existing model transformations (transformation alternatives in our case). After the production of composite transformations, those answering at best to a trade-off among conflicting NFPs must be selected.

Performing a manual selection on a design space of model transformation compositions can be tedious, error-prone and sometimes impossible (i.e., the design space may be infinite). To overcome these issues, some works such as [55; 29; 28; 30], propose the use of Multi-Objective Optimization Techniques (MOOTs) for design space exploration. A MOOT is a decision making process which offers to compare automatically solutions with respect to multiple (most of the time conflicting) objectives (e.g., NFPs); and collect a set of *non-dominated solutions* (also called *Pareto optimal solutions*): given a set of objectives (e.g., NFPs), a solution S1 dominates another solution S2, if S1 is at least as good as S2 in all the objectives, and additionally, S1 is strictly better than S2 in at least one objective.

Yet, using MOOTs to explore model transformation compositions raises new challenges. Firstly, due to some constraints not all possible composition of transformations produce correct architectures. This problem is more detailed in section 4.2. Secondly, the design space grows rapidly with the number of transformations and elements in the model which leads to a combinatorial problem. Thirdly, for reuse and maintenance reasons, model transformations are often structured as chains (i.e., the output model of a transformation becomes the input model of a new transformation). When transformation alternatives exist for each link in such a chain, several exploration links have to be chained, which increase significantly the combinatorial problem. In section 3.3, we detail the second and third problems.

3.2 Identification and selection of architecture alternatives

Software architects need a reliable, rigorous, and automatic process for **selecting** architecture alternatives and ensuring that the decisions made minimize risks and costs; while maximizing profits and optimize NFPs. A good decision-making technique is one that guides the final users towards better, even **optimal** architecture alternatives.

Many programming languages provide different decision making techniques such as the **If-Then-Else statement**. This statement is used to check whether a condition is True or False, and depending on the result select the corresponding solution. An example of the If-Then-Else statement is represented in the following:

```
Program SelectSolution(value) :
// value represent the result of the evaluation of one NFP
    If condition (value >= 0.5) Then
            select the first architecture alternative
    Else
            select the second architecture alternative
    End If
End Program
```

Exemple 3.1: If-Then-Else statement

In addition to that, the functionality of the "If-Then-Else" statement can be extended by adding "Else-If" clauses. These clauses allow the final user to manage more alternatives. This technique is very simple to use and allow selecting good architecture alternatives. For example:

```
Program SelectSolution(value)
// value represent the result of the evaluation of one NFP
    If value >= 0.5 Then
           select the first architecture alternative
    ElseIf value == 0.4 Then
           select the second architecture alternative
    ElseIf value == 0.3 Then
           select the third architecture alternative
    Else
           select the fourth architecture alternative
    End If
End Program
```

Exemple 3.2: Manage more alternatives with Else-if clauses

We can add as many Else-If clauses as we need to provide alternative choices. However, extensive use of the Else-If clauses is laborious. Large systems may admit millions of architecture alternatives; and using Else-If clauses to select the alternatives that optimize the NFPs become very complex, expensive and time-consuming. In order to avoid identifying all the architecture alternatives using Else-If clauses, we proposed to automatically generate them using model transformation compositions techniques.

3.2.1 Automatic enumeration of model transformation alternatives

A model transformation is an automatic process of converting one or multiple models (called *source models*) into one or multiple models (called *target models*). The goal underlying the use of model transformations is to save time and efforts and reduce errors by automating the production and modifications of the models. In this context, we consider model transformation alternatives as transformations that produce different target models when applied on the same source model. These target models represent architecture alternatives which may differ in the NFPs (e.g., reliability, response time, etc.) they exhibit.

However, formalizing architecture alternatives using model transformations is a complex operation: we must identify several transformation alternatives in order to generate all the possible architecture alternatives. To avoid this complexity, we decided to use model transformation composition techniques.

Model transformation composition produce automatically new model transformations (also called **composite transformations**) by combining existing model transformations (in our case transformation alternatives) that (i) are applied on the same source model,

and (ii) produce target models with similar structures, but exhibit different NFPs. Model transformation composition is a generic technique that instead of identifying all the possible transformation alternatives, generate them automatically.

Yet, composing model transformations to formalize architecture alternatives, raises the following problem: do all the combinations of transformation alternatives produce composite transformations generating correct architectures?

Definition 1 *A correct architecture is an architecture that respects a set of structural constraints and conforms to requirements defined by software designers.*

How to ensure (or validate) that composite transformations produce such architectures? This problem is more detailed in the next subsection.

3.2.2 Validation of model transformation compositions

Before explaining how to validate composite transformations, we first give some examples of composite transformations producing valid and invalid architectures. On the left part of figure 3.1, we consider the allocation of software components on hardware components: initially, software components are not deployed on any hardware components. We consider transformations that allocate software components on hardware components. In this example, an allocation is considered correct if it respects the following structural constraints:

1. All the software components must be deployed in order to analyze NFPs.
2. One software component can be deployed on only one hardware component.

On figure 3.1, Output_1 represents the execution of the composite transformation C_1 composed of three transformations T1, T2 and T3: T1 represents the allocation of S1 on H1, T2 the allocation of S1 on H2 and T3 the allocation of S2 on H2. This output model is not valid, since it does not respect the structural constraint 2: the same software component S1 is deployed on two different hardware components at the same time which is impossible. The composite transformation C_3 is also invalid. Once executed C_3 produces an output model which respects the structural constraint 2 but does not respects the structural constraint 1: after applying the transformation, S1 is not allocated to any component. Finally, the composite transformation C_2 is valid; once executed it produces the Output_2 which respects all the predefined constraints: all the software components are deployed only once on the different hardware components.

In the context of model transformations composition, two categories of validation may be used: **a-posteriori** and **a-priori**.

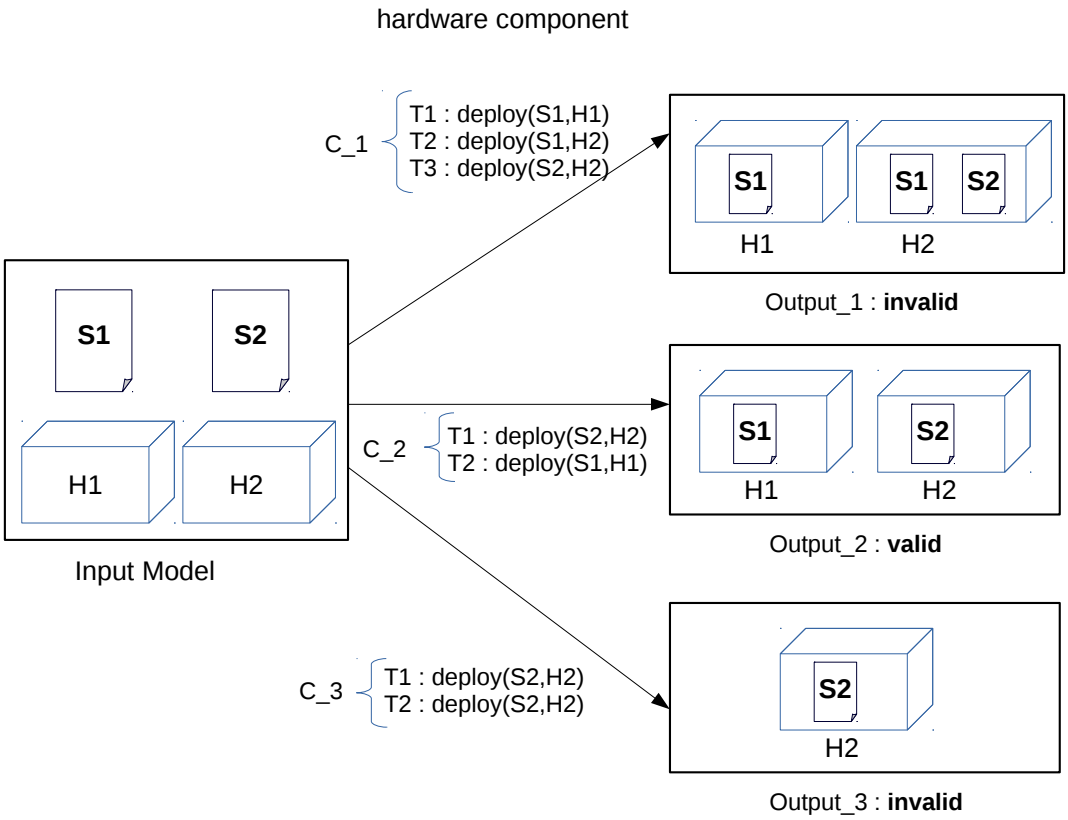

Figure 3.1: Invalid transformations composition

3.2.2.1 A-posteriori validation

The a-posteriori technique consists in executing each composite transformation and check if the resulting target model (i) respects a set of structural constraints; and (ii) contains NFPs conform to some requirements defined by software designers. In figure 3.2, we illustrate the application of the a-posteriori validation technique using the following example. We consider four transformation alternatives ($T1$, $T2$, $T3$, $T4$). Each transformation allows to deploy a software component on a hardware component:

T1 represents the allocation of Sw1 on Hw1,
T2 represents the allocation of Sw1 on Hw2,
T3 represents the allocation of Sw2 on Hw1,
T4 represents the allocation of Sw2 on Hw2,

Composite transformations ($CT_1, CT_2, CT_3, CT_4, CT_5$, ..etc.) are obtained by combining the different transformation alternatives ($T1$, $T2$, $T3$ and $T4$). Afterwards, each composite transformation is executed in order to generate the corresponding target model. Finally,

target models are checked regarding (i) the predefined structural constraints and (ii) the conformity of NFPs.

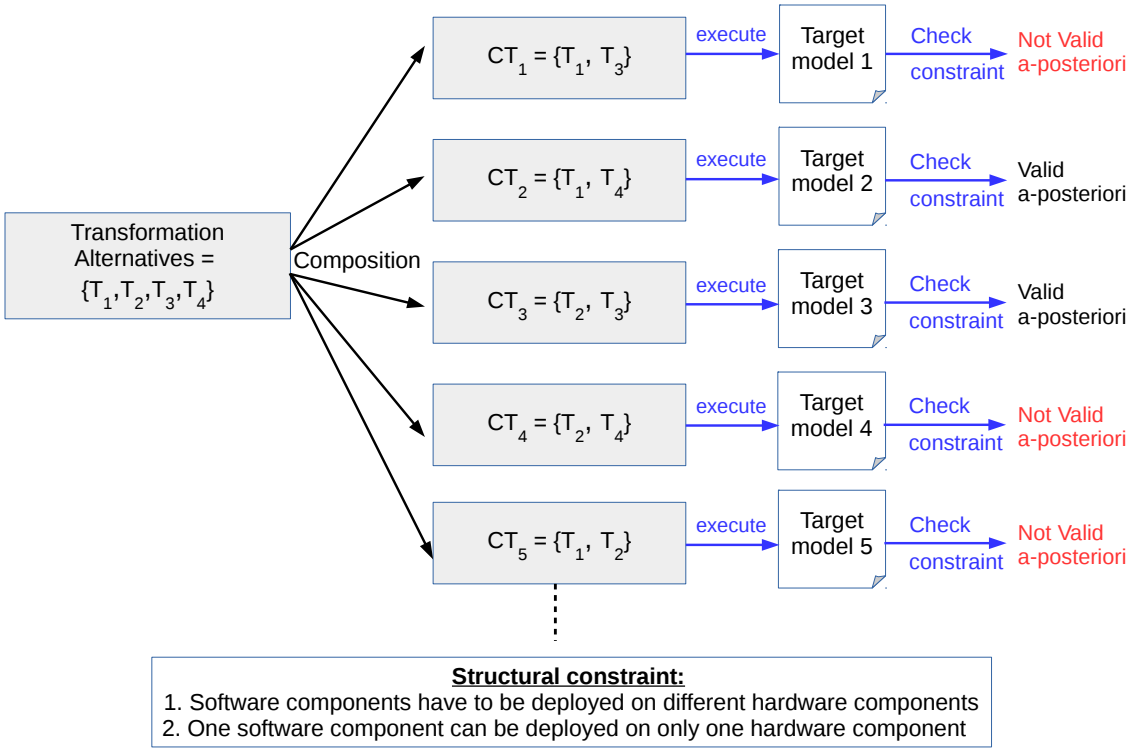

Figure 3.2: a-posteriori validation of composite transformations

This validation technique is easy to apply, it consists only in producing the composite transformations, execute them and check the conformity of the resulting target models. However, this technique is time-consuming and very expensive: we have to systematically execute each composite transformation, and check the validity of the resulting target model. Therefore, it would be better to use a mechanism that validate **a-priori** a composite transformation.

3.2.2.2 A-priori validation

The **a-priori validation** represents the capability of determining at the construction of the composite transformation if it produces a correct architecture or not. Applying the a-priori validation instead of the a-posteriori validation presents the following benefit: reduce the analysis costs by not executing systematically composite transformations and analyzing the resulting architectures. In order to reach this objective, we have to defer the validation of structural constraints at the construction of composite transformations. Consequently the first problem tackled in this thesis is:

Problem1. Ensure (a-priori) that the produced composite transformations always generate architectures that are correct to pre-established constraints.

For NFPs, their evaluation during the production of composite transformations is difficult, and is left as an a-posteriori validation. Thanks to model transformations compositions we can identify automatically all the architecture alternatives, explore the resulting composite transformations and select those answering at best to a trade-off among NFPs.

In some cases (e.g., very large number of transformation alternatives) it would be more interesting to not enumerate all the composite transformations for design space exploration. Instead of that, we use heuristics methods such as multiple objectives optimization techniques to converge quickly to interesting solutions. This approach is more detailed in the next section.

3.3 Exploration of model transformation compositions

Having so far discussed the production of composite transformations, another important aspect of this thesis is the exploration and selection of composite transformations (or also called design space exploration (DSE)). DSE is a central step in any system design. There may be many different composite transformations that implement different architecture alternatives. These different composite transformations have to be explored and evaluated with respect to their objectives (or NFPs) such that a designer can have an overview of architecture alternatives and their levels of quality.

If only a single objective (one NFP) needs to be considered in the DSE, the composites transformations are ordered regarding this objective value. Therefore, there is only one composite transformation that generates the optimal architecture alternative to implement. However, software systems are usually evaluated according to a large variety of objectives (or NFPs) such as reliability, temporal performance, cost, power and energy consumption, size and weight. As these kinds of NFPs are very often conflicting, there is no single optimal composite transformation but a variety of choices that represent different trade-offs. Optimality in this case is defined using the concept of **Pareto-dominance**: a composite transformation dominates another one if it is equal or better in all objectives and strictly better in at least one. In a set of composite transformations, those are called **Pareto-optimal (or non-dominated) solutions**, they represent the composite transformations that are not dominated by any other composite transformation.

Based on these notions, a DSE approach must be characterised as follows:

1. **Search strategy:** It consists in systematically enumerating all possible candidate solutions (composite transformations), evaluating them and identifying the non-

dominated solutions. This strategy is simple to implement, but its cost is proportional to the number of candidate solutions. In addition to that, exploring a large number of solutions leads to an unacceptable exploration time. One way to avoid these problems is to reduce the design space, e.g., using heuristics specific to the treated problem. In this context, **a randomized search strategy** is of great interest. Using this strategy, the design space is searched via specific optimisation approaches. Applying these approaches, new candidate solutions are created using variation operators. Variation operators are stochastic operators which perform random modifications on selected solutions to create new solutions that can be better than the old ones or not. Afterwards, the objectives (e.g., NFPs) of these new candidate solutions are computed, and the results are compared to the old candidate solutions. Based on the comparison results the available information about the design space is increased.

2. **Automatic analysis:** a major step in the exploration of a design space is the analysis of candidate solutions (in our case composite transformations). Indeed, in order to define the set of non-dominated solutions, a comparison operation must be performed among all candidate solutions. A prerequisite for this operation is to analyze the quality of each candidate solution regarding the objectives (in our case the NFPs). Performing the analysis manually is complex and time-consuming. Consequently, DSE approaches should be equipped with analysis techniques that automates the evaluation of candidate solutions regarding all the objectives.

3. **Rapid and effective convergence:** most of the design spaces are composed of too many candidate solutions, and exploring exhaustively such large design spaces is not realistic. Thus, the DSE approach should be able to explore large design spaces at reasonable computational costs.

4. **Precise exploration:** when exploring a design space, some of the candidate solutions may be considered equivalent regarding their impact on all the NFPs, and software designers may be interested in considering only the distinctive candidate solutions. Consequently, the exploration mechanism must provide a method for exploring only interesting candidate solutions and excludes the similar solutions.

In this context, dedicated **Multiple Objectives Optimization techniques (MOOTs)** have been proposed. MOOTs are well known optimization techniques, proposed to (i) deal with the task of simultaneously optimizing two or more conflicting objectives with respect to a set of certain constraints; and (ii) fasten design space exploration using heuristics.

Heuristics allow to reduce considerably the design space, and speed up the process of finding non-dominated solutions. In this thesis, we aim at producing tailored composite transformations to use MOOTs. Thanks to that, not all composite transformations are enumerated, and those answering at best to a trade-off among conflicting NFPs are found in a rather short time.

However, important challenges have to be addressed when applying MOOTs in order to explore and select composite transformations answering at best to a trade-off among NFPs. Firstly, composite transformations are represented by a code written in a programming language. Yet, MOOTs are usually applied on solutions set formalized using specific encoding (binary, real numbers, floating-point, etc.). Thus, we must find a solution to formalize the composition of model transformations in order to use MOOTs. Secondly, the selection of a composite transformation requires to compute the impact of each composition on a set of NFPs. Though, the impact of a model transformation is difficult to formulate a-priori. These two challenges are more detailed in the subsection 3.3.1.

In some cases, to produce target models (or architecture alternatives) with a very low abstraction level, a large number of transformations alternatives have to be used for the same composite transformation. Yet, this transformation presents some drawbacks, including limited reusability and maintainability, reduced scalability and poor separation of concerns. To avoid such issues, MDE introduced the concept of model transformation chains. Indeed, chaining model transformations helps to decompose complex model transformations (in our case composite transformations) into smaller transformations easier to reuse and maintain. In this context, model transformation chains (or MTCs) are used to formalize architecture alternatives. Then, the created MTCs have to be explored and selected in order to find the non-dominated solutions. However, a problem raises when exploring MTCs using MOOTs: how to explore a design space composed of model transformation chains? This problem is more detailed in the subsection 3.3.2.

3.3.1 Structure model transformations for design space exploration

As a first step in any MOOT, the representation of candidate solutions is of great importance. It encodes the appearance, and the behaviour of candidate solutions, which facilitates the application of the different steps of the used MOOT. Formalizing a good representation that is expressive and evolvable is a difficult challenge.

In addition to that, in this thesis, candidate solutions are model transformation compositions. They are usually defined using specific languages (such as ATL, GReAT, Mia-TL, Kermeta etc.). As can be seen in Figure 3.6, the model transformation structure is very

```
// transform classes
getUMLClassesForPackage(umlPackage).each
{
umlClass | var javaClass : JavaClass init JavaClass.new
// ******** creation of the java class from the uml class ********* //
javaClass.name              := umlClass.name
javaClass.belongsTo      := trace_pack2pack.getTargetElement(umlClass.belongsTo)
JavaClass.isPublic         := umlClass.classifierVisibility  == Visibility.public)
trace_class2class.storeTrace(umlClass, javaClass)
JavaPackage.classes.add(javaClass)
}
```

Kermeta transformation rule for uml to java class transformation

```
rule Class2Class {
    from  in  : uml!Class
    to      out : Java!JavaClass
    (
        –------- creation of the java class from the uml class ----
        name       ← in.name,
        isAbstract ← false,
        IsPublic    ← true,
        package    ← in.namspace,
    )
}
```

ATL transformation rule for uml to java class transformation

Figure 3.3: Examples of model transformations representation

expressive: composed of several lines of code, keywords, etc. Thus, it is not trivial to encode such model transformations in order to map them into MOOTs. Consequently the third problem tackled in this thesis is:

Problem2. Structure model transformation compositions in a way that ease their mapping into MOOTs, ensuring that the composite transformations are correct (i.e., produce correct architectures)

To better understand the need to formalize candidate solutions, we take the following example: the application of evolutionary algorithms on candidate solutions. **Evolutionary Algorithms (or EAs)** are well known MOOTs that fasten design space exploration. EAs are inspired from natural evolution. In other words, they are based on populations (i.e., set of candidate solutions of fixed size) that evolve over several generations. In each generation, the EA selects from the population several interesting solutions (also called parent solutions). From these parent solutions, new solutions (called offspring solutions) are created, using **selection** and **variation operators** such crossover and mutation operators. The offspring solutions are then evaluated regarding the objectives and added to the population. These steps are repeated until the most suitable solutions are identified.

The major step of the EAs is the application of variation operators. To perform such operation a *good representation of candidate solutions is essential*. Usually, EAs use binary representations. The most standard one is an array of bits (sequence of 0 and 1, as can be seen in figure 3.4). The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size. This greatly facilitates the application of variation operators.

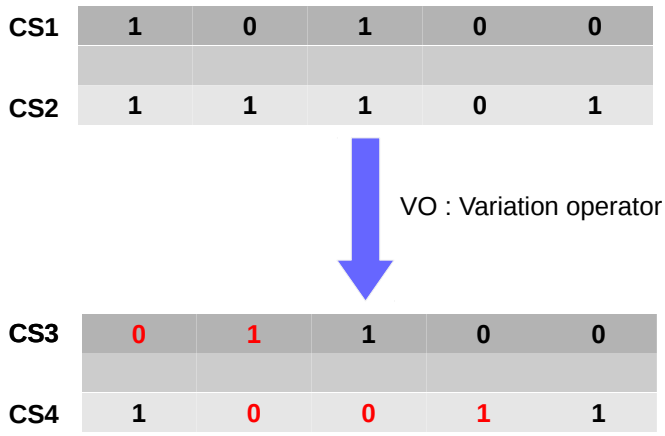

Figure 3.4: Application of variation operators on binary representations

For instance, in figure 3.4, two candidate solutions CS1 and CS2 are defined using binary representations. Consider a variation operator VO that changes bits randomly. Applying VO on CS1 (respectively CS2), creates two new candidates solutions CS3 (respectively CS4). Thanks to this binary formalization, we can easily apply variation operators, which allow to not define and explore all candidate solutions from the beginning. Instead of that, a set of solutions are initially defined, and new candidate solutions are created at different steps of the exploration mechanism to cover as many solutions as possible.

For some design problems, different kind of model transformations (e.g., software component allocations, safety patterns, etc.) can be applied on the source model, which raises the number of model transformation alternatives. Thus, the composition of transformation alternatives becomes very complex to perform. To avoid this complexity, MDE introduced the concept of model transformation chains. It consists in chaining model transformations in order to decompose a complex transformation into smaller transformation links, easier to maintain and reuse. As for composite transformations, we must explore all the created model transformation chains and select those answering at best to a trade-off among conflicting NFPs.

3.3.2 Exploration of chained model transformations

As defined in section 3.2.2, model transformation composition consists in producing a new model transformation called *composite transformation* from a set of existing transformation alternatives that: (i) are applied on the same source model; and (ii) produce different target models with similar structures, but exhibit different NFPs. **Higher Order Transformations** (or HOTs)[56] is a popular way to implement such model transformation compositions.

In some cases, we need to lower the abstraction level of the target models: covering several design problems such as software component allocations, safety patterns, hardware configuration, component selection. To perform that, we must identify several model transformation alternatives, and compose them. Yet, the produced composite transformations present some drawbacks, including limited reusability, reduced scalability, poor separation of concerns, and undesirable sensitivity to changes (limited maintainability).

To resolve these issues, MDE introduced the concept of model transformation chains (or MTCs)[15; 57]. Indeed, chaining model transformations helps to decompose complex model transformations (in our case composite transformations) into smaller model transformations easier to reuse and maintain and more scalable.

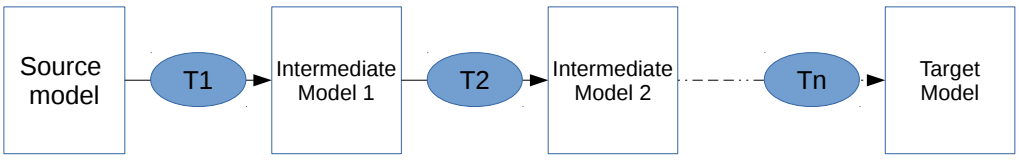

Figure 3.5: Model transformation chain

A model transformation chain is illustrated in Figure 3.5. This chain takes as input the source model defined by the designer of the system, and applies a series of model transformations (also called **links**): T1,T2,...,Tn. Thus, the source model is transformed step by step into intermediate models: each model represents a different level of abstraction. Finally, the last transformation (Tn) produces the last model of the chain which is the target model (or architecture alternative) with the lowest abstraction level.

In our context, the goal of using MTCs is to lower the abstraction level of models progressively in order to evaluate accurately a given system. More precisely, each transformation produces an intermediate model that will be evaluated regarding a specific NFP. However, new challenges arise when evaluating the different links of a model transformation chain: sometimes the intermediate model produced by a link (or a model transformation) of the chain cannot be analyzed because other links impact the result of the analysis.

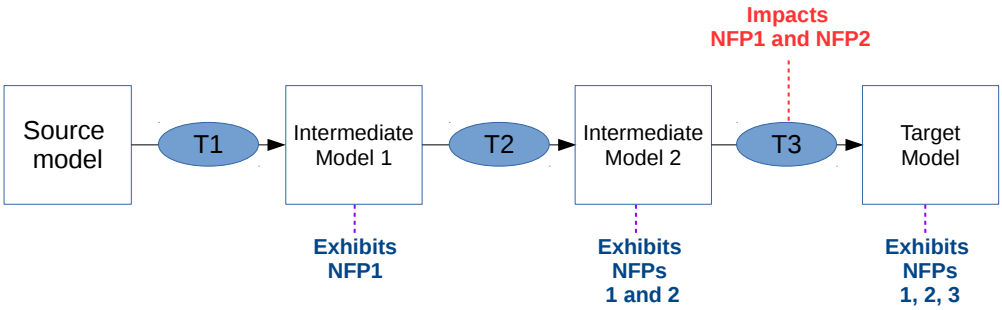

Figure 3.6: NFPs evaluation of model transformation chains

As can be seen in figure 3.6, the transformation $T1$ produces the $intermediate\_model\_1$ that exhibits the $NFP1$, Yet, the result of the evaluation of the $intermediate\_model\_1$ is false, because another transformation $T3$ impacts the $NFP1$ and $T3$ has not yet been executed. That means, we cannot analyze the $intermediate\_model\_1$ without executing the transformation $T3$. Thus, $T1$ and $T3$ are interdependent. The same problem for the analysis of the $intermediate\_model\_2$ that exhibits the $NFP2$. $T3$ impacts $NFP2$, thus $T3$ has to be executed in order to analyse the $NFP2$.

The model transformation chain is only analyzable if all the links of the chain are executed. Thus, the evaluation has to be performed only in the target model: it contains all the necessary data for NFPs analysis. In this context, architectures exploration cannot be done with isolated model transformations, but must consider chains of model transformation alternatives. Consequently the fourth problem tackled in this thesis is:

Problem3. Architectures exploration based on model transformations is a bi-dimensional exploration, both in breadth (composition of transformation alternatives) and in depth (model transformation chains).

3.4 Conclusion

Noticing the difficulties of formalizing architecture alternatives, we decided to use model transformations. More precisely, we formalized architecture alternatives using model transformation alternatives. However, enumerating manually several transformation alternatives to formalize architecture alternatives is very complex. A common solution to avoid such complexity is to use model transformation composition techniques.

Yet, not all model transformation compositions lead to valid architecture alternatives. Thus, model transformation compositions have to be validated. To perform that, we have envisaged two validation techniques: the a-posteriori and the a-priori techniques. We have determined that applying the a-priori validation instead of the a-posteriori validation presents the following benefit: reduce the analysis costs by not executing systematically composite transformations and analyzing the resulting architectures. However using the a-priori validation lead us to tackle our first problem in this thesis:

Problem1. Ensure (a-priori) that the produced composite transformations always generate architectures that are correct to pre-established constraints.

After defining and validating model transformations compositions, we then investigated the importance of using composite transformations in design space exploration. We have determined through this investigation that exploring a design space of composite transformations using heuristics optimization techniques shows several benefits. Firstly, it automates the identification of architecture alternatives answering at best to a trade-off among conflicting NFPs. Secondly, it not enumerates all the composite transformations, but generating them on the fly, which fasten the convergence to the non-dominated solutions. Finally, it offers the possibility to reuse the architecture alternatives in other projects or products by reusing model transformations. However, mixing architecture optimization techniques with model transformation compositions leads us to the second challenge tackled in this thesis:

Problem2. Composite transformations have to be structured in a way that eases their incorporation in optimization techniques.

To lower the abstraction level of target models (architecture alternatives), we investigated the importance of chaining model transformations. We have determined through this investigation that the decomposition of complex model transformations into smaller ones (by chaining) shows several benefits: (i) model transformations are easier to reuse and maintain; and (ii) the evaluation of a given system is more accurate.

However, there are some dependencies among model transformations of a given chain: the intermediate model produced by a transformation of the chain cannot be analyzed

because other transformations impact the result of the analysis. In this case, architectures exploration cannot be done with isolated model transformations, but must consider chains of model transformation alternatives. This leads us to the third challenge tackled in this thesis:

Problem3. Architectures exploration based on model transformations is a bi-dimensional exploration, both in breadth (composition of transformation alternatives) and in depth (model transformation chains).

In the next chapter, we give an overview of the solutions we propose to each of the identified problems and highlight the main contributions of this thesis.

4 Overview of the approach

CONTENTS

4.1 Introduction

As described in the previous chapter, we use model transformation alternatives in order to formalize architecture alternatives. The interest of using model transformation alternatives is that we can compose them to create new model transformation alternatives, and from that automate the identification and generation of architecture alternatives. However, not all composite model transformations generate correct architecture alternatives. Thus, the first contribution of this thesis is to validate a-priori that composite transformations produce correct architecture alternatives (see definition 1). This first contribution is presented in section 4.2.

In this thesis, we aim at exploring a set of architecture alternatives, and selecting only those optimizing a set of conflicting NFPs. Since we formalized architecture alternatives with composite transformations, we perform the optimization process on a design space of composite transformations. Optimizing manually composite transformations can be tedious and error-prone. To overcome this difficulty, we use multiple-objectives optimization techniques (MOOTs) for design space exploration. More precisely, we apply MOOTs on a design space of composite transformations in order to identify the non-dominated composite transformations: composite transformations answering at best to a trade-off among conflicting NFPs. Once executed, the non-dominated composite transformations generate optimal (or near-optimal) architecture alternatives. This second contribution is presented in section 4.3.

For reuse and maintainability reasons model transformations are chained together by providing the output (i.e., the intermediate model) of one transformation as input of another transformation. Thus, to find a set of optimal (or near-optimal) architecture alternatives, non-dominated chained model transformation alternatives must be identified and selected. To identify these alternatives, we rely on the same approach defined in the previous paragraph: applying MOOTs on a design space composed of model transformations chains in order to find the non-dominated ones. Yet, some issues must be solved in order to perform this approach. The intermediate model produced by a link of the chain cannot be evaluated because other links impact the result of the evaluation. To take into account this constraint, we create an extended version of our approach, where model transformation alternatives are identified in each link of the chain, and the evaluation regarding NFPs is only performed on the last link of the chain. This third contribution is presented in section 4.3.

4.2 Formalization of architecture alternatives

The first problem addressed in this thesis emerged in the context of analyzing architecture alternatives regarding a set of conflicting Non-Functional Properties ($NFPs$). Indeed, developing a software that simultaneously improves a set of $NFPs$ is complex to perform. One complexity is partly due to the fact that the $NFPs$ are often in conflicts: improving one particular NFP can degrade other $NFPs$. As a consequence, we have to (i) identify and formalize several architecture alternatives, (ii) analyze each alternative regarding the $NFPs$, and (iii) select the non-dominated architecture alternatives. In the next subsection, we describe how we formalize architecture alternatives in order to (i) ease their analysis regarding NFPs and (ii) facilitate their exploration and selection.

4.2.1 Modelling of architecture alternatives

A common way to formalize architecture alternatives is to use **abstract models**. These models describe an aspect of the system that is not easily or sufficiently captured through system implementation (such as $NFPs$). Thus, using these models, we analyse the $NFPs$ of the system implementation of each architecture alternative.

In order to automate the modelling of architecture alternatives, we use a well-known model driven engineering technique called **model transformations**.

Definition 2 *A model transformation is a software artefact that specifies a set of actions to generate a target model from a source model.*

Definition 3 *In this context, we consider* ***model transformation alternatives*** *are model transformations that generate architecture alternatives when applied to the same source model.*

In this thesis, we focus on model transformation alternatives producing architectures with different NFPs (e.g., reliability, response time, etc.) they exhibit.

Generally, model transformations are written in a model transformation language. In the previous years, several model transformation languages have been proposed to write model transformations such as rule-based model transformation languages, declarative languages. In this thesis, we propose the use of rule-based model transformation languages to write our model transformations.

Definition 4 *A rule-based model transformation is a model transformation made up of a set of transformation rules. A transformation rule specifies the mapping from classes of elements from the source model (e.g., meta-classes of the source meta-model of the transformation), to classes of elements from the target model (e.g., meta-classes of the target meta-model of the transformation).*

More formally, applying rule-based model transformations instantiates for each transformation rule, a set of ***transformation rule instantiations (TRIs)***.

Definition 5 *A transformation rule instantiation TRI_i is the application of a transformation rule on an ordered set of elements from the source model. It can be represented as a tuple $< R, E_i, A_i >$, where:*

1. *R represents the applied transformation rule;*
2. *E_i is i^{th} tuple of elements in the source model that matches the* input pattern *of R;*
3. *A_i is the set of actions that R executes when it is applied to E_i.*

To illustrate this definition of TRI, we introduce the following example: a rule-based model transformation (called $AllocationRule$) made up of one transformation rule which aims to allocate two software components (Sw_1 and Sw_2) on two hardware components Hw_1 and Hw_2.

Applying $AllocationRule$ on the elements of the source model (software and hardware components), instantiates four $TRIs$:

$$
\begin{aligned}
TRI_{11} &=< AllocationRule, \{Sw_1, Hw_1\}, SetAllocation(Sw_1, Hw_1) >, \\
TRI_{12} &=< AllocationRule, \{Sw_1, Hw_2\}, SetAllocation(Sw_1, Hw_2) >, \\
TRI_{21} &=< AllocationRule, \{Sw_2, Hw_1\}, SetAllocation(Sw_2, Hw_1) >, \\
TRI_{22} &=< AllocationRule, \{Sw_2, Hw_2\}, SetAllocation(Sw_2, Hw_2) >.
\end{aligned}
$$

In these $TRIs$, the actions of the rule are noted $SetAllocation(Sw_i, Hw_j)$, which specifies that Sw_i is allocated to Hw_j. For instance, TRI_{11} represents the allocation of $Sw1$ on $Hw1$, and TRI_{22} represents the allocation of $Sw2$ on $Hw2$.

The execution of each TRI transforms a tuple of elements from the source model to their image in the target model. Yet, executing only one TRI is not sufficient to produce an architecture alternative. In this context, we regroup a set of $TRIs$ to form a new model transformation. When executed this transformation should generates an architecture alternative.

Because of the execution semantics of rule-based model transformation languages [58], transformation rule instantiations may be executed in any order without modifying the result of the transformation. This characteristic is called confluence.

Definition 6 *Confluent transformation rule instantiations are rule instantiations that can be applied in parallel or in any order to yield to the same result.*

Taking into account the confluence definition, we must identify sets of confluent $TRIs$, and regroup them to form new model transformations.

The goal underlying the use of rule-based model transformations is to save time and efforts and reduce errors by automating the productions and modifications of the architecture alternatives. However, this technique still lacks the ability to automatically explore complex design spaces. When studying large-scale systems that contain thousands of elements, numerous model transformation alternatives should be developed. The exploration of these alternatives becomes very complex and time-consuming. To cope with this complexity, we identify set of confluent transformation rule instantiations, and compose them in order to automate the development of model transformation alternatives.

4.2.2 Automatic identification of architecture alternatives

Composition of model transformations allow for the production of new model transformations (also called **composite transformation**) by combining existing model transformations (such as confluent transformation rule instantiations in our case). The combination here represents the selection of all or part of a set of confluent transformation rule instantiations instantiations.

In this thesis, we use composite transformations in order to: (i) identify architecture alternatives by combining confluent transformation rule instantiations, and (ii) generate correct architecture alternatives by executing composite transformations.

More formally, we first identify all the transformation rules instantiations ($TRIs$): $T = \{TRI_{11}, TRI_{12}, TRI_{13}, .., TRI_{21}, .., TRI_{ij}\}$. Then, we regroup these $TRIs$ regarding their confluence (see definition 6): $TT = \{CTRI_1, CTRI_2, CTRI_3, .., CTRI_m\}$, where each $CTRI_i$ represents a set of $TRIs$ applied on a set (or tuple) of elements of the source model:

$$
\begin{aligned}
CTRI_1 &= \{TRI_{11}, TRI_{21}, TRI_{31}\},\\
CTRI_2 &= \{TRI_{11}, TRI_{21}, TRI_{32}\},\\
CTRI_3 &= \{TRI_{11}, TRI_{21}, TRI_{33}\},\\
&\ldots\ldots\ldots, CTRI_N.
\end{aligned}
$$

Then, we compose these sets of confluent transformation rule instantiations $CTRIs$ to obtain all the possible composite transformations (CTs):

$$\begin{gathered} CT_1 = \{CTRI_1, CTRI_2\}, \\ CT_2 = \{CTRI_1, CTRI_3\}, \\ CT_3 = \{CTRI_1, CTRI_2, CTRI_3\}, \\ \ldots\ldots\ldots, \\ CT_m = \{CTRI_{n-1}, CTRI_n\}. \end{gathered}$$

Finally, we execute the produced composite transformations to generate the corresponding architecture.

To better understand the composition of sets of confluent transformation rule instantiations, we take the following example: the allocation of two software components ($Sw1, Sw2$) on two hardware components ($Hw1, Hw2$). Initially, we identify four transformation rule instantiations $T = \{TRI_{11}, TRI_{12}, TRI_{21}, TRI_{22}\}$, where each transformation rule instantiation (TRI_{ij}) represents the allocation of one software component (i) on one hardware component (j). For instance, TRI_{11} represents the allocation of Sw_1 on Hw_1, and aT_{12} represents the allocation of Sw_1 on Hw_2.

Then, we regroup the $TRIs$ regarding their confluence (see definition 6). TRI_{11} and TRI_{12} (resp. TRI_{11} and TRI_{12}) are not confluent: applying TRI_{11} with TRI_{12}, we generate a target model where the software component Sw_1 is allocated at the same time on two hardware components. Thus, in this example we have four sets of confluent transformation rule instantiations ($CTRIs$):

$$\begin{gathered} CTRI_1 = \{TRI_{11}, TRI_{21}\} \\ CTRI_2 = \{TRI_{11}, TRI_{22}\} \\ CTRI_3 = \{TRI_{12}, TRI_{21}\} \\ CTRI_4 = \{TRI_{12}, TRI_{22}\} \end{gathered}$$

By composing these $CTRIs$, we produce our composite transformations: $CT_1 = \{CTRI_1, CTRI_2\}$, $CT_2 = \{CTRI_1, CTRI_3\}$, $CT_3 = \{CTRI_1, CTRI_2, CTRI_3\}$,..,$CT_M$. The produced composite transformations represent different allocations of software components on hardware components.

However, all the possible compositions of sets of confluent transformation rule instantiations do not lead to a **valid** composite transformation.

Definition 7 *A valid composite transformation is a transformation that, once executed, produces a correct architecture (see definition 1).*

In the example defined above, assume we have the following structural constraint: *one software component can be allocated on only one hardware component*. Taking into account this constraint, the composite transformation CT_1 is **not valid**. Once executed CT_1 generates an architecture alternative that does not respect the predefined structural constraint: the same software component Sw_1 is allocated on two different hardware components (Hw_1 and Hw_2) at the same time by applying $CTRI_1$ and $CTRI_2$.

In this case, the sets of confluent transformation rule instantiations ($CTRI_1$ and $CTRI_2$) of the composite transformation CT_1 are called **atomic transformation alternatives**.

Definition 8 *Given a set of confluent transformation rule instantiation $CTRI_1$, $CTRI_2$ is an atomic transformation alternative if it contains rules instantiations that are not confluent (see definition 4.2.2) with rules instantiations of $CTRI_1$. In this case, $CTRI_1$ and $CTRI_2$ cannot be applied to the source model in parallel, and either $CTRI_1$ or $CTRI_2$ should be applied.*

Thus, in order to generate correct architectural design solutions, composite transformations must be validated by identifying atomic transformation alternatives. The validation of composite transformations is described in the next subsection.

4.2.3 Validation of composite transformations

To ensure that composite transformations generate correct architectures, we use two validation techniques: **a-posteriori** and **a-priori**. More precisely, we use an a-posteriori validation to evaluate the impact of composite transformations on a set of $NFPs$, and an a-priori validation to construct composite transformations.

4.2.3.1 A-posteriori validation

The a-posteriori validation consists in executing a composite transformation, evaluating the impact of these composites on NFPs, and checking whether the resulting target model (i) respects a set of structural constraints; and (ii) exhibits NFPs conform to some requirements defined by software designers. Although the a-posteriori validation has a great advantage to validate composite transformations, it also has drawbacks. This validation is time-consuming and very expensive: it requires to systematically execute each composite transformation, evaluate the resulting target model regarding the NFPs, and check the

correctness of the target model. Therefore, it would be better to use a mechanism that validates **a-priori** composite transformations.

In this thesis, NFPs evaluation is quantitative. Thus we must compute the impact of model transformations on the NFPs. Yet, this impact is difficult to compute a-priori. Consequently, we use the a-posteriori validation to evaluate NFPs. On the other hand, checking that structural constraints are respected is qualitative, and to perform that, we use the a-priori validation. This validation technique is described in the next paragraphs.

4.2.3.2 A-priori validation

The a-priori validation consists in the capability of determining at the construction of composite transformations whether they generate architectural solutions that respect a set of structural constraints. More precisely, to create valid composite transformations, we have to transfer the structural constraints expressed on architectural design solutions into constraints on the model transformation composition mechanism.

Structural constraints are usually expressed using dedicated languages such as the Object Constraint Language (OCL) [59]. The OCL is already largely used to express constraints on models. It is a declarative language that provides rules to specify structural constraints on the instances of a given model. Once OCL constraints are expressed on the meta-models, we can check whether the models respect the constraints or not.

However, in this thesis we aim at expressing structural constraints on model transformations (developed using a specific language such as ATL), not on abstract models. It is difficult to express OCL structural constraints on model transformation alternatives. Instead of that, we adopt the following solution.

First, remember that we aim at producing composite transformations by **selection** of confluent transformation rule instantiations from a set of all the identified transformation rule instantiations T. To check structural constraints at the creation of composite transformations, we constrain the selection of transformation rule instantiations with logical expressions (also called boolean formulas) while taking into account the confluence property. More formally, we use the following points:

1. $T = \{TRI_{11}, TRI_{12}, .., TRI_{ij}\}$, the sets of transformation rule instantiations.
2. TRI_i a transformation rule instantiation of T.
3. T_i a subset of T, where TRI_i is not listed ($T - \{TRI_i\}$).
4. The following function
   $Select : \quad T \quad \rightarrow \mathbb{B} = \{True, False\}$

$TRI \rightarrow b$, where b is $True$ if TRI should be selected to be part of a composite transformation, and $False$ if TRI should be excluded from a composite transformation.

5. *BoolExpr*: a boolean expression over sets of transformation rule instantiations. It uses the (i) *Select* function, and (ii) simple boolean operators *and*, *or*, and *not* ($\wedge$, $\vee$, and $\neg$). For instance, a $BoolExpr_1$ is: $[Select(TRI_{11}) \vee Select(TRI_{12})]$. Assume in this example that TRI_{11} allocates a software component Sw_1 on a hardware component Hw_1 and TRI_{12} allocates the same software component Sw_1 on a different hardware component Hw_2. In this example, $BoolExpr_1$ defines the confluence property: we must select either TRI_{11} or TRI_{12} for the allocation.

Based on these points, we define a boolean formula called *StructuralConstraints* which express structural constraints on transformation rule instantiations (*TRIs*):

$$\textbf{StructuralConstraints} = \bigwedge_{\mathbf{i=1}}^{\mathbf{N}} (\textbf{Select}(\textbf{TRI}_\mathbf{i}) \Rightarrow \textbf{BoolExpr}(\textbf{T}_\mathbf{i})). \tag{4.1}$$

In equation 9.1, the boolean formula *StructuralConstraints* express for each transformation rule instantiation TRI_i the transformation rule instantiations that can be composed with TRI_i (included) and those which cannot be composed with $CTRI_i$ (excluded).

For instance, consider T a set of four transformation rule instantiations: $T = \{TRI_{11}, TRI_{12}, TRI_{22}, TRI_{21}\}$. Each TRI allocates a software component on a hardware component. In addition, we have the following structural constraint: *different software components cannot be allocated on the same hardware components*. This structural constraint may be expressed as a conjunction of the following boolean expressions:

$$\textbf{StructuralConstraints} =$$
$$\begin{array}{l} Select(TRI_{11}) \Rightarrow [Select(TRI_{22}) \wedge \neg Select(TRI_{21}) \wedge \neg Select(TRI_{12})] \\ \bigwedge\ Select(TRI_{12}) \Rightarrow [Select(TRI_{21}) \wedge \neg Select(TRI_{22}) \wedge \neg Select(TRI_{11})]. \end{array}$$

In this example, selecting TRI_{11} implies to select TRI_{22}, **and** not select both TRI_{12} and TRI_{21}. Selecting TRI_{11} with TRI_{12} results in an invalid composite transformation that violates the confluence characteristic: the software component Sw_1 is allocated on two hardware components Hw_1 and Hw_2. In addition to that, when selecting TRI_{11}, we can select only TRI_{22}. Selecting TRI_{21}, we produce a composite transformation that violates the structural constraint defined before: the software components Sw_1 and Sw_2 are

allocated on the same hardware component Hw_1. The same for the application of the transformation rule instantiation TRI_{12}.

A valid composite transformation is thus defined by a function *Select* that satisfies equation 9.1. Given this function, atomic transformation alternatives are identified and applied on the input model. In order to find the *Select* function, we decide to use SAT solving techniques [60]. These techniques allow us to automatically (i) evaluate the existence of solutions (atomic transformation alternatives) to equation 9.1, and (ii) extract such solutions.

This concludes the presentation of our solution to solving the first problem tackled in this thesis: formalize architecture alternatives by composing atomic transformation alternatives; validate a-priori the resulting composite transformations; and evaluate a-posteriori NFPs of composite transformations. The second problem tackled in this thesis, is the exploration of composite transformation and the selection of the non-dominated ones. We describe this solution in the next section.

4.3 Design space exploration

The second problem addressed in this thesis emerged in the context of exploring architecture alternatives, and selecting the optimal (or near optimal) ones. Since we automate the modelling of architecture alternatives using composite transformations, the exploration and selection process is performed on a design space of composite transformations.

When the number of possible composite transformations is very large, it is difficult to identify all of them, evaluate them and select the non-dominated composite transformations: it leads to an unacceptable exploration time. Instead of that, we use heuristics methods such as evolutionary algorithms (EAs) to converge quickly to non-dominated composite transformations. This approach is described in the next subsection.

4.3.1 Exploration of model transformation compositions

In order to explore a design space of composite transformations, we define an exploration engine based on EAs principles. Figure 9.1 illustrates the four steps of the exploration engine.

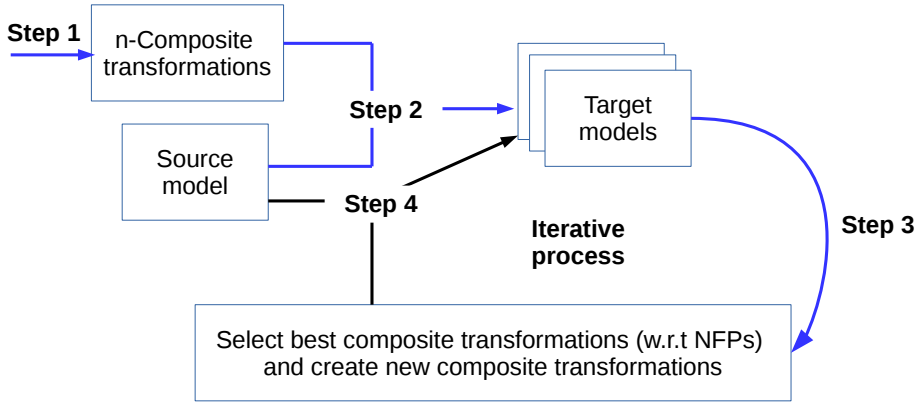

Figure 4.1: Overview of the exploration engine

4.3.1.1 Step 1

EA usually starts from an initial population of randomly generated genomes (composite transformations in our case), where each genome is composed of a set of genes (atomic transformation alternatives in our case). This initial population has a fixed size of composite transformations n (n can be set, for instance, by end-users of our exploration engine).

4.3.1.2 Step 2

From the source model, and the initial population of composite transformations, we produce n target models. More precisely, we execute each composite transformation to generate the corresponding target model. The generated target models are then analyzed regarding the NFPs. Of course, they are analysed with respect to the set of NFPs of interest for the system. The results of this analysis are then used to select the best composite transformations with respect to NFPs, and from that to create new composite transformations.

4.3.1.3 Step 3

To select and create composite transformations, we rely on EAs principles: selection and genetic operators. In our case, the selection operator consists in finding the best composite transformations regarding the NFPs evaluation (see step 2).

Then the selected composite transformations are used to create new composite transformations. More precisely, the creation of these new composite transformations is performed using the mutation and crossover operators (also called **genetic operators**). The crossover operator consists in taking more than one genome and producing a new genome

from them. Many crossover operators exist in the literature. For instance, the two-point crossover operator aims to create new genomes from existing ones by (i) choosing randomly two cutting points in the existing genomes organism (the genes); and (ii) exchanging the genes between the two points. A two-point crossover operation is illustrated on figure 4.2: two binary solutions, S_1 and S_2, are cut between the two positions (3,4) and (7,8) to produce two new solutions S'_1 and S'_2.

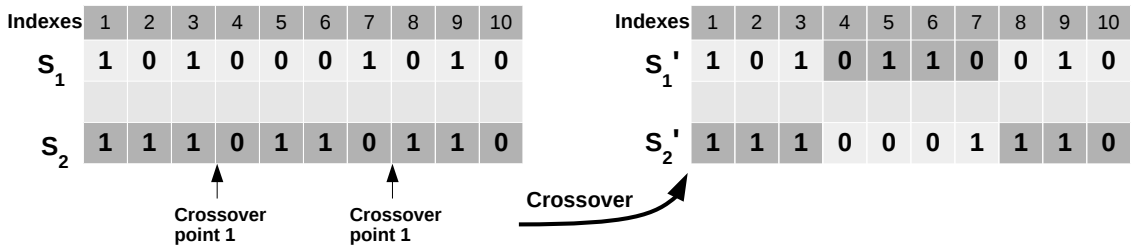

Figure 4.2: Two-point crossover operator applied on binary solutions

When applying genetic operators on any type of genomes, we must take into account an important constraint: *genomes on which we apply the genetic operators must have the same size*. We respect this constraint by fixing the size of composite transformations: all the composite transformations have the same number m of atomic transformation alternatives (aTs): $CT_i = \{aTa_j\}_{j=1,..,m}$.

In addition to that, we organize composite transformations as arrays. Arrays are useful critters in order to group a set of variables of the same type (atomic transformation alternatives in our case) that can be accessed by a numerical index. As can bee seen in figure 9.2 the index 3 contains the atomic transformation alternative aTa_{31}. Finally, we give a specific order for the atomic transformation alternatives in the arrays: atomic transformations alternatives are placed on the same index in different arrays. For instance, in figure 9.2 aTa_{31} and aTa_{32} are placed on the same index 3. This encoding facilitates the application of genetic operators: cutting, exchanging elements and mutating can be done by choosing randomly indexes of the arrays.

Figure 9.2 illustrates the application of the two-point crossover operator on composite transformations. In this example, when we apply the a-priori validation to produce the initial composite transformations, we identify the atomic transformation alternatives: aTa_{i1} and aTi_{i2}. Then, these alternatives are place in the same index in the arrays (composite transformations). For instance, aTi_{51} cannot be composed with aTi_{52}. Thus, aTi_{51} and aTi_{52} are alternatives and they are placed on the same index in different arrays.

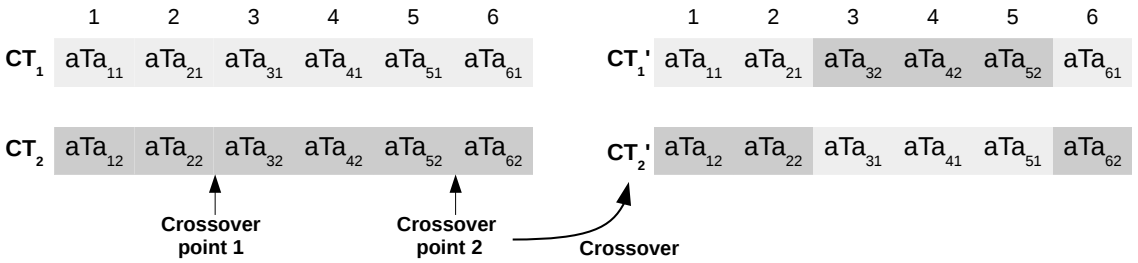

Figure 4.3: The application of genetic operators on composite transformations

Applying the crossover operator, two valid composite transformations, CT_1 and CT_2, are cut between the two points (2,3) and (5,6) to produce two new composite transformations CT_1' and CT_2'. Thanks to our encoding, we do not have to check the validity of CT_1' and CT_2'. Since we ordered (with the indexes of the arrays) the atomic transformation alternatives of the composite transformations regarding the ability to compose them, we can exchange them and produce new valid composite transformations without repeating the a-priori validation step. On figure 9.2, aTa_{31} (resp. aTa_{41} and aTa_{51}) and aTa_{32} (resp. aTa_{42} and aTa_{52}) are placed on the same index in the arrays, and exchanging them we ensure that the new composite transformations are valid.

Note here the interest of having atomic transformation alternatives as ordered sets, of fixed size. Composite transformations resulting from a genetic operator (e.g., crossover) preserve the same characteristic to form new composite transformations. Thus, we ensure that the new produced composite transformations are valid without applying the the a-priori validation step.

4.3.1.4 Step 4

Since EAs are iterative processes, we iterate on steps 2 and 3 until a convergence criteria is met. Common convergence criteria are:

1. A fixed number of iterations reached.
2. Allocated budget (computation time/money) reached
3. The highest solutions regarding the evaluation no longer produce better results.

In our exploration engine, we use the first and third convergence criteria in order to identify the non-dominated composite transformations.

Actually, we use model transformation chains in order to formalize architectures alternatives. Thus, to identify architecture alternatives answering at best to a trade-off among conflicting NFPs, we have to explore a design space composed of model transformation chains. This extended version of our exploration engine is described in the next subsection.

4.3.2 Exploration of model transformation chains

To find non-dominated architecture alternatives, we formalize architecture alternatives using model transformation chains. Remember that a model transformation chain is composed of different links, each link represents a model transformation once executed generates an intermediate model, and the last link generates the target model.

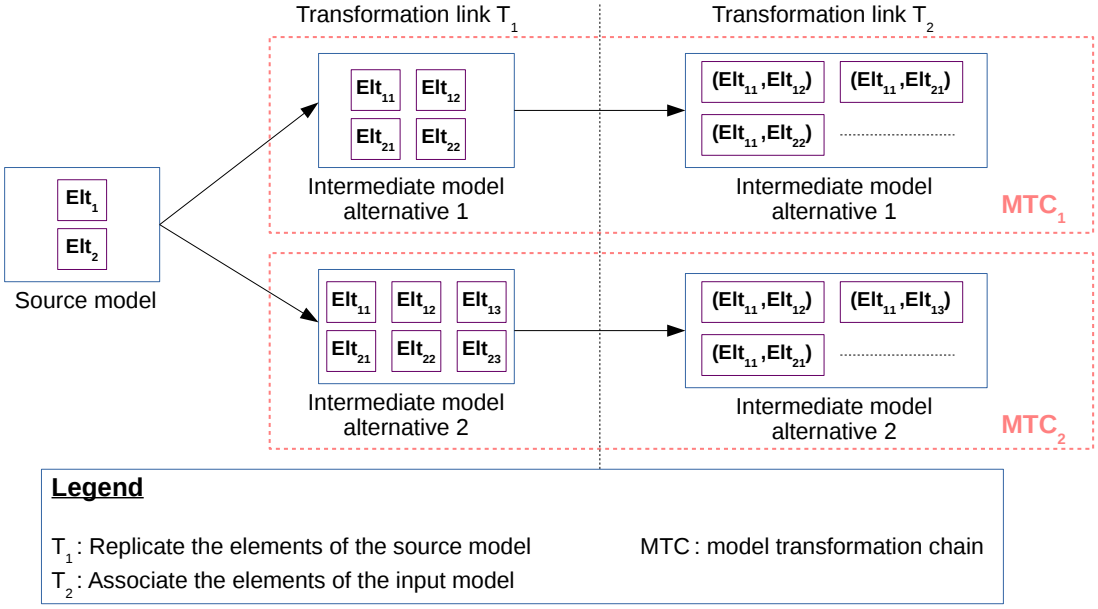

Figure 4.4: Overview of model transformation chains

As can be seen in figure 9.3, the intermediate model of each link of the chain can be composed of a different number of elements (different sizes). In figure 9.3, applying the transformation link T_1, we replicate a certain number of times the elements of the source model. Since we have different alternatives (duplicate, triplicate, etc), each link produces its own set of intermediate model alternatives. These alternatives will be used as input of the following link, and produce another set of alternative models with different sizes. In figure 9.3, applying the transformation link T_2, we associate the replicated elements and produce intermediate models with different sizes. Thus, the size of the intermediate model of each link depends on the output of the previous links of the chain.

Once we define model transformation chains, we must identify the set of non-dominated ones. To find this set we (i) apply EAs selection and genetic operators on these chains, and (ii) select the non-dominated model transformation chains. To develop this approach, we rely on our exploration engine (defined in subsection 4.3.1). However, difficult sub-problems have to be solved in order to apply our exploration engine on model transformation chains. In the following we precise these sub-problems, and give our propositions to solve them:

1. In order to find non dominated model transformation chains, we evaluate each link of the chain regarding a set of conflicting NFPs. Our first sub-problem is that *the intermediate model produced by a link of the chain cannot be analyzed because other links impact the result of the analysis*. In this case, the exploration cannot be done with isolated links, but we must consider chains of model transformation alternatives when evaluating the impact on NFPs. To solve this first sub-problem we perform the following solution. For each link we define model transformation alternatives that will be chained to the transformation alternatives of the following links, and we evaluate the target model generated by the final link.

2. The design space contains model transformation chains (genomes) having a different number of atomic transformation alternatives (genes). As defined in subsection 4.3.1, genomes on which we apply EAs genetic operators must have the same size (same number of genes). To overcome this second sub-problem, we apply the genetic operators on the chained model transformation alternatives, instead of applying them on the atomic transformation alternatives of a given model transformation alternative (as we performed in the step 3 of the subsection 4.3.1).

Figure 9.4 gives an overview of the process we propose to explore a model transformation chain: from the definition of a chain, having for each link a small set of transformation alternatives, we first produce an intermediate model that results from the composition of these alternatives. This first step is highlighted with bullet 1 on the figure 9.4, and repeated for each link of the transformation chain until the target model is produced.

We assume, that intermediate models do not contain all the information required to analyze $NFPs$ of the architecture alternatives: intermediate models are enriched in each link of the transformation chain. Once produced, the target model is analyzed with respect to NFPs (see bullet 2 on figure 9.4), and the analysis results are used to evaluate composite transformations.

The process we propose is iterative: each iteration produces, executes, and evaluates a sub-chain of composite transformations. In addition, because of the combinatorial com-

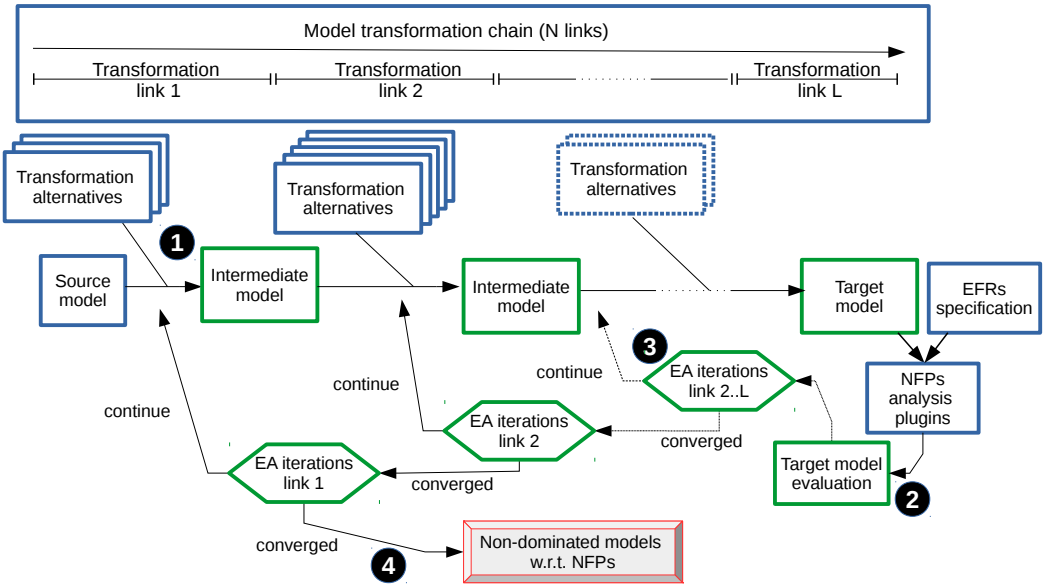

Figure 4.5: Approach Overview

plexity of the design space exploration, it is not possible to enumerate, execute, and evaluate all the composite transformations. As a consequence, we rely on evolutionary algorithms (EAs) to implement this iterative exploration (see bullet 3 on figure 9.4). In addition, we can see on figure 9.4 that the proposed process is made up of embedded loops, each loop being dedicated to explore composite transformations of a given link in the transformation chain. When an inner loop has converged, other transformation candidates may be evaluated for the outer loop, thus producing a new intermediate model for the inner loop. The convergence criteria for each loop relies on convergence criteria of EAs and is parameterized by an end-user of our approach.

When the top-level loop has converged, the exploration process stops and returns the set of non-dominated models that were found (see bullet 4 in figure 9.4).

4.4 Conclusion

In this chapter we have given an overview of the contributions of this thesis. The first set of contributions addresses the first problem tackled by this thesis which is formalizing and producing correct architecture alternatives. We perform the formalization of architecture alternatives using rule-based model transformations. Indeed, these model transformations propose useful mechanisms such as confluence, transformation rule instantiations

and composition techniques that help us to automate the formalization, the identification and also the production of architecture alternatives.

However, not all composite transformations generate correct architectures. To tackle this challenge, we (i) validate a-priori composite transformations using boolean expressions that enforce the respect of predefined structural constraints and SAT solvers; and (ii) evaluate the conformity of NFPs, using an a-posteriori validation. Once produced, composite transformations are executed on a source model in order to generate and evaluate architecture alternatives.

After the validation of composite transformations, we tackle the second challenge of this thesis which is to identify and select architecture alternatives answering at best to a trade-off among conflicting NFPs. Since we formalize architecture alternatives using composite transformations, the identification and selection steps are applied on a design space of composite transformations. In order to face the problems raised by architecture exploration, and in particular the combinatory explosion of composite transformations, we use evolutionary algorithms to converge quickly to non-dominated composite transformations. This required to structure composite transformations in order to guarantee that the application of the selection and genetic operators produces new valid composite transformations. More precisely, we structure composite transformations as ordered sets of atomic transformation alternatives, with a fixed size. With this solution, new composite transformations resulting from the application of genetic operators necessarily produce correct architectural design solutions.

For reuse and maintainance reasons, model transformations are often structured as chains (i.e., the output model of a transformation becomes the input model of a new model transformation). Thus, we formalize architecture alternatives as chained model transformation alternatives. Consequently, to identify optimal (or near optimal) architecture alternatives, we have to explore a design space composed of chained model transformation alternatives. To reach this objective, we apply EAs steps on a model transformation chain. We iterate over links of a model transformation chain and produce the most suitable composite transformations for each link. Once model transformation alternatives are defined, we evaluate their impact on NFPs, select the best alternatives regarding the evaluation results, and apply genetic operators on them to create new chained model transformation alternatives. We repeat these operations until we produce non-dominated chained model transformation alternatives.

The next chapters detail our proposals. First, to formalize architecture alternatives we propose the use of rule-based transformation languages (such as ATL). These languages propose mechanisms that ease the reuse, maintainability and composition of model trans-

formations. This first proposal is detailed in Chapter 5. Then, the exploration of composite transformations and model transformation chains using evolutionary algorithms is detailed in Chapter 6. Finally, the experimental validation and assessment of all our contributions are presented in Chapter 7.

5 Identification and composition of model transformation alternatives

CONTENTS

5.1 Introduction

In this chapter we describe how we formalize architecture alternatives using model transformation techniques. Firstly, in section 5.2, we present the technical solutions we use to automate the identification and production of architecture alternatives using rule-based model transformations. More precisely, we show how to identify, and compose confluent transformation rule instantiations. Secondly, in section 5.3, we describe our solution to ensure a priori that composite transformations generate correct architecture alternatives. Finally, in section 5.4 we explain and validate our contributions with a well-known rule-based model transformation language: the ATL language.

5.2 Formalization of confluent transformation rule instantiations

As defined in the previous chapters, we rely on rule-based model transformations (see definition 4) in order to formalize architecture alternatives. More precisely, executing rule-based model transformations, we produce transformation rule instantiations (*TRIs*) (see definition 5). These *TRIs* can be composed to create new model transformations, and from that automate the identification and generation of architecture alternatives. However, not all compositions of *TRIs* lead to correct architecture alternatives (see definition 1): some constraints (confluence and/or structural) have to be checked in the composition of *TRIs* to generate correct architecture alternatives. In the following subsections we describe the execution of rule-based model transformations, and the composition of *TRIs* by taking into account confluence (see definition 6) and structural constraints.

5.2.1 Execution of rule-based model transformations

Figure 5.1 illustrates the generation of architecture alternatives from the execution of a rule-based model transformation. Initially, software architects develop a source model with a high abstraction level representing the system to study. Then, a rule-based model transformation composed of a set of transformation rules (see bullet 1 in figure 5.1) is defined. These transformation rules implement different design patterns (e.g., software components allocation, model refinement, etc.).

Secondly, the defined rule-based model transformation is executed, and is instancied for each transformation rule, a set of **transformation rule instantiations (*TRIs*)** (see bullet 2 in figure 5.1). Each *TRI* represents an application of the transformation rule

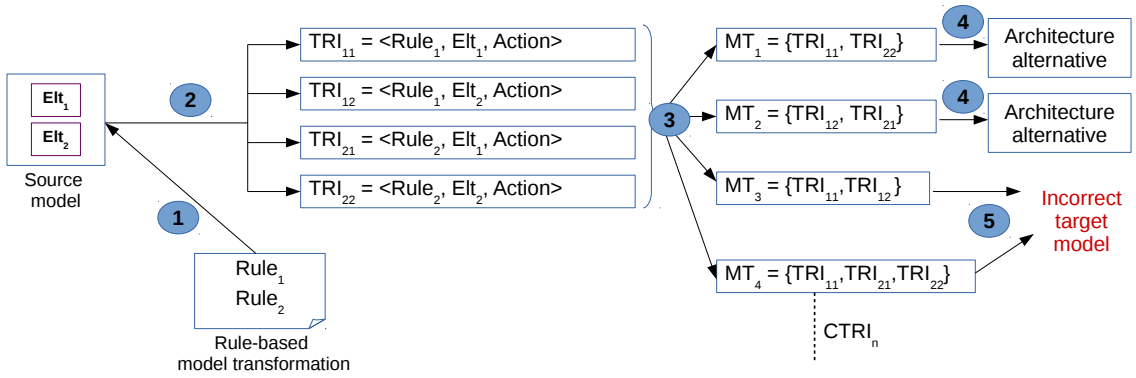

Figure 5.1: Overview of the execution of rule-based model transformation rules

which transforms elements from the source model into their image in the target model (see definition 5).

Finally, the *TRI* are regrouped together to form new model transformations. When executed these new transformation generate the corresponding architecture alternatives. (see bullets 3 and 4 in figure 5.1).

These steps represent a good scenario where executing a rule-based model transformation on a source model generates architecture alternatives. However, not all the generated architecture alternatives are correct (see bullet 5 in figure 5.1). Because, additional constraints have to be checked on these alternatives in order to validate them. Applying an a posteriori validation requires to systematically execute each new model transformation, and check the correctness of the architecture alternative. This validation mechanism is time-consuming. Therefore, we use a mechanism that validates a priori that the new model transformations generate correct architecture alternatives. More precisely, we check at the construction of the new model transformations, that (i) the selected *TRIs* can be applied together (i.e., they are confluent), and (ii) the selection of *TRIs* produce composite transformations that generate correct architecture alternatives (i.e., respect some structural constraints). In the following subsections we first present the identification of confluent *TRIs*.

5.2.2 Identification of alternative transformation rule instantiations

Because of the confluence characteristic, not all *TRIs* can be selected to be applied together. Thus, we must identify confluent and not confluent *TRIs*. In the remainder of this thesis, we use the notation $TRI_i \oplus TRI_j$ to represent that two transformation rule instantiations TRI_i and TRI_j are **not confluent**. In other words, these two transformation rule

instantiations are exclusive and represent **alternatives**. In our approach, we encode the detection of **non-confluence** according to two types of situation:

1. An alternative exists when more than one transformation rule can be applied to the same set of elements in the source model. Formally, this means:

$$\begin{aligned} \exists(R,R')\ s.t.\ R \neq R' and\ TRI_i =< R,E_i,A_i > \\ and\ TRI_j =< R',E_j,A_j >\ and\ E_i = E_j \end{aligned} \tag{5.1}$$

   For instance, consider a rule-based model transformation called "$safetyPattern$" aiming at replicating a certain number of times software components. This model transformation is composed of two transformation rules: $R = duplicateComponent$" which duplicates software components and $R' = triplicateComponent$" which triplicates software components. In this example applying R on a software component Sw_i and applying R' on the same software Sw_i are alternatives.

2. An alternative exists when a rule is applied to set of tuples but the action this rule performs should be applied for a subset of these tuples. We propose to identify such situations by annotating a rule with a description of its **unicity scope**: given a tuple of elements in a TRI, the unicity scope (US) of this TRI identifies elements of the tuple for which the rule can be applied only once. More formally, non-confluence exists when:

$$\begin{aligned} \exists(E_i,E_j)\ s.t. E_i \neq E_j\ and\ TRI_i =< R,E_i,A_i > \\ and\ TRI_j =< R,E_j,A_j >\ and\ US(TRI_i) = US(TRI_j) \end{aligned} \tag{5.2}$$

   For instance, consider the transformation rule $AllocationRule$ which allocates software components (Sw_1, Sw_2) on hardware components (Hw_1 and Hw_2). In this transformation software components must be allocated on one and only one hardware component. Thus, the unicity scope of the $TRIs$ are software components: $\mathbf{US}(TRI_{11} = \{AllocationRule, \{Sw_1, Hw_1\}, Allocate\}) = \{Sw_1\}$, and $\mathbf{US}(TRI_{12} = \{AllocationRule, \{Sw_1, Hw_2\}, Allocate\}) = \{Sw_1\}$. Thanks to this unicity scope we can detect alternative TRIs: TRI_{11} and TRI_{12} are alternatives because $US(TRI_{11})$ = $US(TRI_{12})$ = $\{Sw_1\}$.

Once we identify **alternative transformation rule instantiations**, we (i) regroup them into sets, (ii) extract from each set one TRI, and (iii) form new sets composed of only confluent $TRIs$. Once composed these sets of confluent $TRIs$ produce new model transformations that generate architecture alternatives.

More formally, we first put all alternative $TRIs$ in the same set. In the following we call such sets **Set of Alternative Transformation Rule Instantiations** (S), where $S = \{TRI_i\}_{i=1..P}$:

$$\forall i, j \in [1,P], i \neq j, TRI_i \oplus TRI_j \tag{5.3}$$

Then, we encode sets of confluent transformation rule instantiations as follows. Given n sets of alternatives transformation rules instantiations: $(S_1, S_2, .., S_n)$ we identify sets of confluent transformation rule instantiation ($CTRIs$) by performing the cartesian product of these sets of alternatives transformation rules instantiations:

$$CTRIs = S_1 \times S_2 \times .. \times S_n = \{(s_1, s_2, .., s_n) \mid s_1 \in S_1 \textit{ and } s_2 \in S_2 \textit{ and } s_n \in S_n\} \tag{5.4}$$

We illustrate these notions with the following example: a source model made up of two software components (Sw_1, Sw_2) and two hardware components (Hw_1,Hw_2), and the transformation rule *AllocationRule* that allocates software components on hardware components. Applying the transformation rule *AllocationRule* on the described source model, we produced the following $TRIs$:

$$\begin{aligned}
TRI_{11} &= < AllocationRule, \{Sw_1, Hw_1\}, SetAllocation(Sw_1, Hw_1) >, \\
TRI_{12} &= < AllocationRule, \{Sw_1, Hw_2\}, SetAllocation(Sw_1, Hw_2) >, \\
TRI_{21} &= < AllocationRule, \{Sw_2, Hw_1\}, SetAllocation(Sw_2, Hw_1) >, \\
TRI_{22} &= < AllocationRule, \{Sw_2, Hw_2\}, SetAllocation(Sw_2, Hw_2) >.
\end{aligned}$$

Two applications of the transformation rule *AllocationRule* are considered to be alternatives if they both allocate the same software component (e.g., Sw_1) on different hardware components. The software component, in each TRI, defines the unicity scope: the $AllocationRule$ cannot be applied more than once on to the same software component. In this example, TRI_{11} is an alternative of TRI_{12} and TRI_{21} is an alternative of TRI_{22}. In other words, this example leads to two sets of alternative transformation rule instantiations:

$$\begin{aligned}
S_1 &= \{TRI_{11}, TRI_{12}\} \\
S_2 &= \{TRI_{21}, TRI_{22}\}
\end{aligned}$$

By performing the cartesian product of S_1 and S_2, we obtain our sets of confluent transformation rule instantiations ($CTRIs$):

$$CTRI_1 = \{TRI_{11}, TRI_{21}\}$$
$$CTRI_2 = \{TRI_{11}, TRI_{22}\}$$
$$CTRI_3 = \{TRI_{12}, TRI_{21}\}$$
$$CTRI_4 = \{TRI_{12}, TRI_{22}\}$$

Finally, we compose these confluent transformation rule instantiations to produce new model transformations (*composite transformations*) that generates architecture alternatives. However, not all composition of sets of confluent transformation rule instantiations lead to valid composite model transformations. In addition to the confluence of $TRIs$, we have to take into account structural constraints defined by system designers. The solution we propose to check these structural constraints is detailed in the next section.

5.3 Composition of atomic model transformation alternatives

Once executed, composite transformations must generate correct architecture alternatives (see definition 1). According to the definition 1, to check the correctness of architecture alternatives we must check the satisfiability of these alternatives regarding a set of structural constraints. As stated before, architecture alternatives are models, and to check the satisfiability of a model regarding some structural constraints a solution is to (i) embed structural constraints on models, and (ii) apply a logical reasoning on these models. The most used solution to perform these steps is the Object Constraint Language (OCL). OCL is a textual language based on a mathematical reasoning that allows to specify and verify additional constraints on models.

Yet, in this thesis we aim at expressing structural constraints on rule-based model transformations, not on models. To perform that, we transfer OCL structural constraints into constraints expressed on the selection of transformation rule instantiations ($TRIs$). This solution is detailed in the following paragraphs.

First, remember that we aim at producing composite transformations by performing the cartesian product of all the identified sets of alternatives transformation rules instantiations. To validate a priori such composite transformations, we constrain the selection process in order to ensure that the resulting architecture respects predefined structural constraints. To perform that, we formalize constraints on the selection of $TRIs$ with logical

expressions. Then, we introduce these logical expressions in a Boolean Satisfiability problem (SAT), and automatically extract validity preserving sets of *TRIs* (also called **atomic transformation alternatives**) using a SAT solver.

Note that the formalization using boolean expressions is totally generic: it is proposed independently of a specific architecture optimization problem. We explain this contribution in the following subsection.

5.3.1 Constraints Expressed on rule-based model transformations

In order to express structural constraints on the selection of *TRIs*, we use the following equation:

$$StructuralConstraints = \bigwedge_{i=1}^{N} (Select(TRI_i) \Rightarrow BoolExpr(T_i)) \tag{5.5}$$

This equation rely on the following points (defined in section 4.2.3.2):

1. $T = \{TRI_i\}_{i=1..N}$, a set of transformation rule instantiations.
2. TRI_i a transformation rule instantiation of T.
3. T_i a subset of T, where TRI_i is not listed in $(T - \{TRI_i\})$.
4. The function *Select* applied on *TRIs*:
   $Select: \quad T \quad \rightarrow \mathbb{B} = \{True, False\}$
   $\quad\quad\quad\quad TRI \rightarrow b$, where b is $True$ if TRI should be included in the composite transformation, and $False$ if TRI should be excluded from the composite transformation.
5. *BoolExpr*: a boolean expression over the set of *TRIs*. It uses the (i) *Select* function, and (ii) simple boolean operators *and*, *or*, and *not* ($\wedge$, $\vee$, and $\neg$). This expression formulates the interactions between *TRIs*. For instance, TRI_j and TRI_k are alternatives (see subsection 5.2.2) and they cannot be applied together. *BoolExpr* in this case is equal to: $\neg(Select(TRI_j) \wedge Select(TRI_k))$.

In equation 5.5, selecting each TRI_i implies to select a subset T_j of *TRIs* from T, and exclude another subset $(T - T_j)$ of *TRIs*. Note that, the selection and exclusion of *TRIs* is determined thanks to the boolean expression *BoolExpr*. In the following *StructuralConstraint*: $Select(TRI_i) \Rightarrow [Select(TRI_j) \wedge \neg Select(TRI_l)]$, the application of TRI_i enforces the selection of TRI_j and the exclusion of TRI_l.

For example, consider T a set of four $TRIs$: $TRI_{11}, TRI_{21}, TRI_{31}, TRI_{41}$. The structural constraints on the selection of these $TRIs$ is expressed as a conjunction of the following boolean expressions:

- $Select(TRI_{11}) \Rightarrow (Select(TRI_{21}) \wedge \neg Select(TRI_{31})) \vee (\neg Select(TRI_{21}) \wedge Select(TRI_{31}))$;
- $Select(TRI_{41}) \Rightarrow \neg(Select(TRI_{11}))$;
- $Select(TRI_{21}) \Rightarrow TRUE$;
- $Select(TRI_{31}) \Rightarrow TRUE$.

In this example, selecting TRI_{11} implies to select either TRI_{21} or TRI_{31} (but not both). In addition selecting TRI_{41} implies to exclude the application of TRI_{11}. Finally, selecting TRI_{21} or TRI_{31} has no further implications.

Once structural constraints on the selection of $TRIs$ are expressed, we can identify atomic transformation alternatives (see definition 8). The identification step is explained in the following subsection.

5.3.2 Formalization of the transformation rule instantiations selection

Considering the definitions given above, for each set of alternative transformation rule instantiations ($S = \{TRI_i\}_{i=1..P}$), the identification of atomic transformation alternatives may be decomposed into the following formulas:

1) AtLeastOne, dedicated to ensure that at least one of the alternative $TRIs$ is selected:

$$\mathbf{AtLeastOne}(\mathbf{S}) = \bigvee_{i=1}^{P} Select(TRI_i) \tag{5.6}$$

2) AtMostOne, dedicated to ensure that only one of the alternative $TRIs$ is selected (otherwise alternatives would still exist in the atomic transformation alternative):

$$\mathbf{AtMostOne}(\mathbf{S}) = \bigwedge_{i=1,j=1,i\neq j}^{i=P,j=P} \neg(Select(TRI_i) \wedge Select(TRI_j)) \tag{5.7}$$

Combining equations (5.6) and (5.7), we obtain *SelectOne*, dedicated to select exactly one TRI from the set of alternative transformation rule instantiations S:

$$\mathbf{SelectOne}(\mathbf{S}) = AtLeastOne(S) \wedge AtMostOne(S) \tag{5.8}$$

3) StructuralConstraints, as defined in equation (5.5).

Finally, for each set of alternative transformation rule instantiations (S_i), the selection of a valid set of *TRIs* boils to evaluate the satisfiability of the boolean formula:

$$\bigwedge_{\mathbf{i=1}}^{\mathbf{P}} (\mathbf{SelectOne(S_i)}) \wedge \mathbf{StructuralConstraints} \tag{5.9}$$

Transformation alternatives are thus defined by a function *Select* that satisfies equation (5.9). Given this function, the selected *TRIs* are applied on the source model, and the others are excluded. In this thesis, we use SAT solving techniques, in order to (i) evaluate the existence of solutions (validity preserving *TRIs*) to equation (5.9), and (ii) extract such solutions.

5.3.3 Automatic extraction of validity preserving transformation rule instantiations

In the previous subsection, we defined a boolean formula 5.9 to select sets of validity-preserving *TRIs* and from that produce transformation alternatives. More precisely, we perform this selection process by determining if there exists an assignment of values (TRUE/FALSE) to the function *Select* in the boolean formula 5.9 such that the formula evaluates to true. To determine assignment values for each function *Select* in the boolean formula 5.9 we rely on SAT solving techniques.

Given a boolean formula, SAT solving techniques find whether there is assignments of values to boolean variables in the formula such that it evaluates to True. In the past, SAT has been used to solve different problems such as planning problems, routing of FPGAs, scheduling problems. To the best of our knowledge, SAT has not been used in a problem including model transformations, and applying these techniques to automate the identification of atomic transformation alternatives represents an originality of our work. In general solving a problem using SAT involves two steps: **modelling** and **solving**.

5.3.3.1 Modelling

This step involves formulating the problem as a boolean (or logic) formula. A boolean formula consists of (i) a set of atomic propositions such as boolean variables (i.e., these variable can have two value True or False); and (ii) a set of logical operators such as $\Rightarrow, \Leftrightarrow, \neg, \vee, \wedge$.

In order to apply a SAT solving technique on a boolean formula a prerequisite is to translate the formula in a Conjunctive Normal Form (CNF). CNF is a standard format

accepted by most SAT solvers. A formula is said to be in the CNF, if it is a conjunction of disjunctions. Given a boolean formula β translated in the CNF, the boolean satisfiability problem is to determine whether there is an assignment of values to all the variables that makes β true.

In this thesis, we formulate our problem using the equation (5.9). This equation is a boolean formula, and we have to transform this equation into the CNF. Since the equation (5.9) is based on the implication (⇒) operator, we translate the equation by using the following instruction: $A => B$ is equivalent to $\neg A \vee B$.

To illustrate the modelling step of SAT, we take the following example. Consider the following boolean formula (β):

- $\beta = [Select(TRI_{11}) \Rightarrow (Select(TRI_{22}) \wedge Select(TRI_{32}))] \wedge [Select(TRI_{12}) \Rightarrow (Select(TRI_{21}) \wedge Select(TRI_{31}))]$

Applying the instruction defined above, this expression is translated into a CNF:

- $\beta_1 = [\neg Select(TRI_{11}) \vee (Select(TRI_{22}) \wedge Select(TRI_{32}))] \wedge [\neg Select(TRI_{12}) \vee (Select(TRI_{21}) \wedge Select(TRI_{31}))]$;
- $\beta_2 = [\neg Select(TRI_{11}) \vee Select(TRI_{22})] \bigwedge [\neg Select(TRI_{11}) \vee Select(TRI_{32})] \bigwedge [\neg Select(TRI_{12}) \vee Select(TRI_{21})] \bigwedge [\neg Select(TRI_{12}) \vee Select(TRI_{31})]$.

Then, we determine whether there is an assignment to all the variables ($Select(TRI_{ij})$) that makes this expression true in order to find our atomic transformation alternatives: $\{Select(TRI_{11}) = false,\ Select(TRI_{21}) = true,\ Select(TRI_{22}) = true,\ Select(TRI_{31}) = true,\ Select(TRI_{32}) = true\}$.

Once the problem is modelled in a CNF format, it can be solved quite quickly using the second step of SAT: solving.

5.3.3.2 Solving

Given a boolean formula β translated in the CNF, the **solving** step of SAT finds whether there is a single assignment for all the boolean variables of β that makes the expression true. If an assignment is found, then β is **satisfiable**, otherwise β is **unsatisfiable**. To perform that, SAT solvers provide a generic combinatorial reasoning and search platform. More precisely, these solvers use Davis-Putnam-Logemann-Loveland (DPLL) [61; 62] algorithm or a variant to solve SAT problems. In this thesis, we decide to use the well known SAT solver called Sat4j [63].

To evaluate the existence of solutions to equation (5.9), we translate the equation in a CNF. Then, to extract such solutions we apply the solver SAT4j on the translated equation. To illustrate this last step we take the same example defined in the modelling step. After the translation of the boolean expression in a CNF, we then apply the solving step. More precisely, we use the produced CNF as input to the Sat4j solver. Applying Sat4j, we produce a set of solutions satisfying the boolean formula defined in Equation 5.9. These solutions represent atomic transformation alternatives (aTa):

$$aTa_1 = \{TRI_{11}\};$$
$$aTa_2 = \{TRI_{22}\};$$
$$aTa_3 = \{TRI_{32}\};$$
$$aTa_4 = \{TRI_{22}, TRI_{32}\};$$
$$...$$

Thus, for each boolean formula (in the example above we have only one boolean formula) we identify a set of atomic transformation alternatives, and compose them to generate correct architecture alternatives.

To summarize with our composition mechanism (that selects a set of $TRIs$ among alternatives), a composite transformation is considered to be valid if (i) its $TRIs$ are confluent (as defined in subsection 5.2.2), and (ii) structural constraints among transformation rule instantiations are satisfied (as defined in subsection 5.3.1).

In the next section, we describe the languages that we used to prototype and validate the contributions defined before.

5.4 Specification of transformation rule alternatives with ATL

In order to illustrate our explanations, prototype and validate our approach we rely on two main languages: the Atlas Transformation Language (ATL) [8] and the Transformation Rules Catalog (TRC). The ATL is used to develop our rule-based model transformations. On the other hand the TRC is an internal language used to specify structural constraints on rule-based model transformations. Using ATL and TRC together allows us to identify confluent and not confluent $TRIs$, express structural constraints on $TRIs$, and provide data

for the solver to identify transformation alternatives. In the following section we first describe the ATL language.

5.4.1 ATL overview

In this thesis, we use ATL as an implementation language to prototype and validate our approach. Any hybrid transformation language with the same semantics (rule-based, pattern-matching triggered) could have been used to prototype our approach.

The ATLAS Transformation Language (ATL)[8] was introduced by the Atlas Group and the TNI Valiosys Company in 2003. As such it provides a transformation engine (based on declarative and imperative languages) able to transform any given source model to a specified target model.

The ATL transformation language is based on **rules** that are either matched in a declarative way or called in an imperative way. The execution is primarily triggered by **matched rules**. Thus, an ATL transformation consists of a set of declarative *matched rules*, each specifying a *source pattern* (the **"from"** section) and a *target pattern* (the **"to"** section). The source pattern is made up of (i) a set of objects identifiers, typed with metaclasses from the source metamodel and (ii) an optional OCL [59] constraint acting as a *guard* of the rule. The target pattern is a set of objects of the target metamodel and a set of *bindings* that assign values to the attributes and references of the output objects.

Besides matched rules, ATL also provides so called helpers that are similar to e.g., Java methods in declarative style. Helpers make it possible to define factorized ATL code that can be called from different points of an ATL matched rule.

To illustrate an ATL transformation we use the example define in 5.2. On the right part of figure 5.2, we have two metamodels *UML* and *JAVA*. *UML* defines two meta classes: *Class* which contains two attributes (name and namespace), and *Attribute* which also contains two attributes (name and type). On the other hand, *JAVA* defines two meta classes: *JavaClass* which contains four attributes (name, isAbstract, package and isPublic) and *Field* which contains five attributes (name, isStatic, isPublic, isFinal and type). We now define an ATL transformation program that transforms any given UML Class into Java Class and UML Attribute into Java Field.

The code that transforms these simple models is depicted on the left part of figure 5.2. We have two matched rules named Class2Class and Attr2Field. The first matched rule *"Class2Class"* consists in transforming each UML *Class* into a JAVA *Class*, while the second matched rule *"Attr2Field"* consists in transforming each UML *Attribute* into a JAVA *Field*. To perform that, each matched rule matches elements from the source model in its

```
rule Class2Class {
      from  c  : UML!Class
      to      out : JAVA!JavaClass
      (
            name              ← c.name,
            isAbstract        ← false,
            isPublic          ← c.isPublic(),
            package           ← c.namespace,
      )
}

helper context UML!ModelElement def :
            isPublic() : Boolean =
                  self.visibility = #public ;

rule Attr2Field {
      from  a  : UML!Attribute
      to      f : JAVA!Field
      (
            name      ← a.name,
            isStatic  ← true,
            isPublic  ← true,
            isFinal   ← false,
            owner     ← a.owner,
            type      ← a.type,
      )
}
```

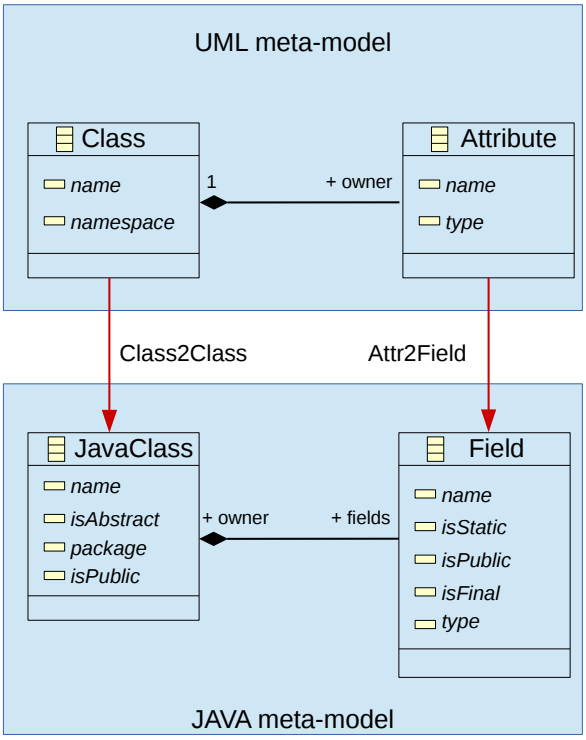

Figure 5.2: Example of an ATL transformation

source pattern ("from part") and transforms them into elements from the target model with its target pattern ("to part").

More precisely, the source pattern of the matched rule *Class2Class* specifies that the rule should be executed for all objects *{c}* (source object identifiers) of type *Class* (defined in UML ECORE meta-model). The target pattern of this rule specifies that, when this rule is executed, one object is created in the target model: *out* of type *JavaClass*.

On the other hand, the source pattern of the second matched rule *Attr2Field* specifies that the rule should be executed for all objects *{a}* of type *Attribute* (defined in UML ECORE meta-model). The target pattern of this rule specifies that, when this rule is executed, one object is created in the target model: *f* of type *Field*.

In this thesis, we use ATL to implement recurrent design patterns such software components allocation, safety design patterns and model refinements (e.g., code generation). Listing 5.1 provides an overview of the ATL code used to implement the transformation rule (*m_AllocationRule_MultiCore*) which allocates software components on multi-core hardware components.

In Listing 5.1, the source pattern specifies that the *m_AllocationRule_MultiCore* rule (*m_AllocationRule_MultiCore*) should be executed for all pair of objects *{Sw,Hw}* (source

object identifiers) of type *Component* (defined in MM ECORE meta-model) such that *Sw* is a software component and *Hw* is a multi-core hardware component (as stated by the OCL constraints in lines 6, 8) and 10). Notice that the OCL constraint defined in line 10 is a helper used to check if the hardware component is multi-core or not.

The target pattern of the *m_AllocationRule_MultiCore* rule specifies that, when this rule is executed, one object is created in the target model: *pa* of type *PropertyAssociation*. This property association is meant to represent in the target model, the allocation of *Sw* on *Hw*. *setAllocation* is an ATL helper that initializes the property value in order to model that *Sw* is allocated to *Hw*.

Exemple 5.1: ATL matched rule for components allocation rule

```
1  rule m_AllocationRule_MultiCore {
2    -- pattern matching section
3   from
4    Sw:MM!Component,
5    Hw:MM!Component (
6     Sw.category = #software
7     and
8     Hw.category = #hardware
9     and
10    Hw.isMulticore()
11   )
12   -- creation section
13  to
14   pa:MM!PropertyAssociation (
15    name<-"allocate_software_on_hardware",
16    value<-thisModule.setAllocation(Sw,Hw)
17   )
18 }
19
20 helper context def: setAllocation : Sequence(MM!PropertyAssociation) =
21         select(Sw|Sw.category = #software)
22             -> collect (Sw | getAllComponents()
23                 -> select(Hw | Hw.category = #hardware)
24                     -> collect (Hw|
25                         thisModule.resolveMatchedTupleIfExists
26                             (Sequence{Sw,Hw}, 'allocate'))
27                 )
28
29
30 helper context MM!Component def : isMulticore() : Boolean =
31     if(self.category = #hardware and self.hasProperty('isCore')) then
32         self.getModalPropertyValue('isCore').ownedValue.value
33     else
34         if(self.category = #core) then
35             self.eContainer().isMulticore()
36         else
37             false
38         endif
39     endif;
```

Given the execution semantics of ATL, the *m_AllocationRule_MultiCore* rule produces for each tuple {software,hardware} of the source model, a transformation rule instantiation $TRI_{ij} =< m\_AllocationRule\_MultiCore, \{Sw_i, Hw_j\}, setAllocation(Sw_i, Hw_j) >$. In these TRIs, the actions of the rule are noted $SetAllocation(Sw_i, Hw_j)$, which represents the creation of a property association specifying that Sw_i is allocated to Hw_j. For instance, $TRI_{11} =< m\_AllocationRule\_MultiCore, \{Sw_1, Hw_1\}, setAllocation(Sw_1, Hw_1) >$ represents the allocation of Sw_1 on Hw_1, and
$TRI_{21} =< m\_AllocationRule\_MultiCore, \{Sw_2, Hw_1\}, setAllocation(Sw_2, Hw_1) >$ represents the allocation of Sw_2 on Hw_1.

Yet, applying only ATL on a given source model we produce *TRIs* without taking into account possible structural constraints. To produce *TRIs* that respect such constraints, we rely on (i) the pattern matching semantics of ATL, and (ii) an internal language called **Transformation Rules Catalog** (TRC).

More precisely, we use the TRC to express structural constraints on ATL transformations rules. Then, we execute the pattern matching semantics of ATL to produce *TRIs*. Finally, from these produced *TRIs* and by instantiating the structural constraints of the TRC, we identify validity preserving *TRIs*, and produce transformation alternatives.

In the following section, we present in details the TRC.

5.4.2 Transformation Rules Catalog (TRC)

Expressing structural constraints on a large number of ATL transformation rules is complex to perform and time-consuming. This complexity is due to the fact that we have to express (using a boolean expression) for each transformation rule instantiation TRI_i the *TRIs* that can be applied together with TRI_i and the *TRIs* that cannot be applied with TRI_i. Instead of expressing manually structural constraints on *TRIs*, we use the **Transformation Rules Catalog (TRC)**.

The TRC is an internal language which identifies all the transformation rules to apply on a given source model, and provides a way to formalize structural constraints on the application/instantiation of ATL transformation rules. The TRC is made up of the following parts:

1. A description of the applied model transformation alternatives. A list of **modules** and **transformation** rules being part of each transformation alternative.

2. Structural constraints on the selection of transformation rules, i.e., a set of boolean constraints using OCL queries to retrieve elements on which rules should be ap-

plied or excluded. Exploiting these OCL queries, we generate the boolean formulas (see equation (5.9)) aiming at selecting validity preserving *TRIs* and from that (by applying a solver SAT) produce atomic transformation alternatives.

3. **Helpers** helping to define factorized ATL code that can be called when expressing structural constraints on ATL transformation rules.

```
1  Modules
2  {
3      Allocation_Problem_multicore.atl : m_AllocationRule_MultiCore;
4      Allocation_Problem_not_multicore.atl : m_AllocationRule_NotMultiCore;
5  }
6  Alternatives
7  {
8      Allocation_MultiCore : { Allocation_Problem_multicore.atl };
9      Allocation_NotMultiCore : { Allocation_Problem_not_multicore.atl };
10 }
11 Structural_Constraints
12 {
13     //MultiCore allocation
14     Apply(Allocation_MultiCore.m_AllocationRule_MultiCore, {Sw, Hw})
15     [(excludes (Allocation_MultiCore.m_AllocationRule_MultiCore,
16             {getNotCollocated})
17         and     requires (Allocation_MultiCore.m_AllocationRule_MultiCore,
18             {getCollocated}))];
19     //Not MultiCore allocation
20     Apply(Allocation_NotMultiCore.Allocation_Problem_not_multicore, {Sw, Hw})
21     [(excludes (Allocation_NotMultiCore.Allocation_Problem_not_multicore,
22             {getNotCollocated})
23         and     requires (Allocation_NotMultiCore.Allocation_Problem_not_multicore,
24             {getCollocated}))];
25 }
26 Helpers //ATL helpers
27 {
28     helper list getCollocated:
29     {  Sw.ownedPropertyAssociation
30         ->any(pa|pa.property.name='Collocated').ownedValue
31             ->asSequence()->at(1).ownedValue";
32     };
33     helper list getNotCollocated:
34     {  Sw.ownedPropertyAssociation
35             ->any(pa|pa.property.name='Not_Collocated').ownedValue
36                 ->asSequence()->at(1).ownedValue";
37     };
38 }
```

Exemple 5.2: An overview of the Transformation Rules Catalog

Listing 5.2 provides a subset of the TRC we used to describe transformation alternatives for the design pattern: software component allocation. This TRC is made up of the following parts:

1. In the listing, lines 1 to 11 provide the list of ATL modules. We have two modules in this example: *Allocation_Problem_multicore.atl* and *Allocation_Problem_not_multicore.atl*. These modules represent alternatives of the allocation of software components on hardware components.
   The *Allocation_MultiCore* alternative is made up of transformation rules described in the previous section 5.4.1: the transformation rule *m_AllocationRule_MultiCore*. The *Allocation_NotMultiCore* alternative is made up of very similar transformation rule: *m_AllocationRule_NotMultiCore*. This transformation rule allocates software components on mono-core hardware components.

2. In this example, we assume that software components have different levels of criticality. In addition to that, we have the following structural constraint to fulfil: "Software components with different level of criticality must be allocated on different hardware components". This structural constraint is expressed in lines 13 to 23: when applying the ATL transformation rule *Allocation_Problem_multicore* to a pair of software/hardware components, *(Sw, Hw)*, then this rule should (i) **not be applied** to pairs of elements *(Sw', Hw)* if Sw and Sw' should not be collocated, and (ii) **applied** to pairs of elements *(Sw", H)* if Sw and Sw" should be collocated. Very similar structural constraints are expressed for the application of the *Allocation_Problem_not_multicore* alternative in lines 26 to 34.

3. In the listing, lines 38 to 52 provide two helpers *getCollocated* (resp. *getNotCollocated*) which aim to retrieve a set of components that should be collocated (resp. not be collocated) with *Sw*.

Once we define ATL transformation rules and the TRC, we can identify validity preserving transformation rule instantiations *TRIs*, and by applying the SAT solver produce atomic transformation alternatives. In the following subsection, we explain how we extract such transformations.

5.4.3 Extraction of Atomic Transformation Alternatives

In the previous section we mentioned that the identification of validity preserving *TRIs*, imply the utilisation of ATL, the TRC, a solver SAT and of course a source model.

More precisely, from a source model representing a given embedded system, we first write our rule-based model transformations using ATL: a set of matched rules implementing design patterns (such as safety design patterns). Then, we construct a TRC containing

the structural constraints provided by the system designer. on our ATL matched rules (as presented in subsection 5.4.2)

Once, ATL transformation rules and the TRC are defined, we first execute the ATL transformation rules. More precisely, the pattern matching of each matched rule is executed on the source model. This execution is performed using the ATL execution runtime: EMFTVM [64]. This transformation engine iterates the set of matched rules, and search in the source model for the elements that match the constraint guard of the matched rules (see subsection 5.4.1 for an example of a constraint guard).

However, executing ATL matched rules, we can only produce target models and not *TRIs*. Thus, we provide some modifications in our framework in order to capture *TRIs*. More precisely, in our framework, we iterate the set of matched rules, searches on the source model for the elements that match the constraint guard of the executed matched rules, and structure theses elements on data representing our *TRIs*. Notice that in this step we do not produce target models, but only *TRIs*.

From the identified *TRIs*, and by applying the predefined TRC, we construct the boolean equations (see equation 5.9) that formalize structural constraints on *TRIs*. More precisely, we (i) iterate on the *TRIs*, (ii) extract from these *TRIs* the applicable matched rules, (iii) search in the TRC the transformation rules that match the extracted data, and (vi) produce the corresponding boolean equation.

Once all the boolean equations are identified, the solver SAT4j is executed to identify a set of atomic transformation alternatives.

5.5 Conclusion

In this chapter we have detailed the first contribution of our work: the a priori validation of composite transformations. The two main steps in this contribution are (i) the composition of rule-based model transformations to identify and generate automatically architecture alternatives, and (ii) the satisfaction of structural constraints on rule-based model transformation to enforce the production of valid composite transformations.

Firstly, we relied on rule-based model transformations to formalize architecture alternatives. Once executed, transformation matched rules define a set transformation rule instantiation (*TRIs*). Using these *TRIs* and basing on the confluence characteristic, we automate the identification of confluent transformation instantiations. Once we produce confluent transformation instantiations, the main challenge is to ensure that their composition generate valid composite transformations. For this challenge, we (i) express the structural constraints on *TRIs* as logical expressions using the TRC and ATL, and (ii) use

a SAT solving technique (SAT4j) to compute transformation alternatives that satisfy the defined logical expression. Finally, atomic transformation alternatives are composed to produce valid composite transformations.

Thanks to these definitions and formalizations, we can ensure that when executed composite transformations generate correct architecture alternatives. However, an important step now is to select composite transformations answering at best to a trade-off among conflicting *NFPs*. Enumerating and selecting manually composite transformations is impractical, and to avoid that we perform the selection process using heuristic methods (such as evolutionary algorithms). Our solution is described in the next chapter 6.

6 Multi-Objective selection and composition of composite transformations

CONTENTS

6.1 Introduction

In this chapter we describe how we explore a design space of composite and chained model transformations using evolutionary algorithms (EAs).

Firstly, in section 6.2, we describe how we explore a design space of composite transformations using EAs. EAs are based on selection and genetic operators. In order to apply these operators on composite transformations while guaranteeing structural constraints, a proper encoding must be defined. Our encoding of composite transformations is defined in subsection 6.2.1. Once composite transformations are well-encoded we can select the best composites regarding the NFPs results (described in subsection 6.2.2), and apply genetic operators (described in subsection 6.2.3) on the selected composite transformations. We use a well known EA called NSGA-II to identify the non-dominated composite transformations. The application of NSGA-II is presented in subsection 6.2.4.

Secondly, in section 6.3, we consider model transformations structured as chains, use EAs to explore these chains and select the non-dominated ones. Yet, some issues have to be solved in order to apply EAs on model transformation chains. The intermediate model generated by a link of a model transformation chain cannot be analysed regarding a set of *NFPs* because (i) other links impact the result of the analysis; or (ii) the intermediate model does not contain all the necessary information for analysis. In addition, each link of a model transformation chain produces an unknown number of atomic transformation alternatives for the next link of the chain. Applying EA genetic operators on solutions with an unknown number of elements is difficult to perform. To solve these issues, we split our previous process into two steps. The first one applies the EA selection and genetic operators on the links of a model transformation chain. The second step performs the global analysis of the chain. These steps are detailed in section 6.3.

6.2 Exploration of composite transformations using EAs

Exploration of composite transformations is a difficult task, mainly because of (i) the fast growth of the number of composite transformations, and (ii) existing conflicts among NFPs impacted by these transformations.

To address similar problems, optimisation methods based on **evolutionary algorithms (EAs)** have been proposed. Inspired from genetics, EAs are based on the following principles:

1. An initial population of genomes with a fixed size (i.e., the size is fixed by the user of the EA) is produced randomly. Where each genome is composed of a set of genes.

2. Then, the best genomes are selected regarding a set of objective functions (e.g., NFPs in our case).

3. A new population of genomes called offspring population is produced using a selection operation and genetic operators (i.e., mutation, crossover).

4. The new genomes are evaluated regarding the objective functions to determine their quality.

5. Finally, the steps above are repeated until a set of non-dominated genomes is determined: given a set of objectives, a genome G_1 dominates another genome G_2, if and only if the following conditions are true: (i) the genome G_1 is no worse than G_2 in all objective functions; and (ii) G_1 is is strictly better that G_2 in at least one objective function.

Applying EAs principles on a search space of composite transformations we identified the following issues. Firstly, composite transformations are sets of atomic transformation alternatives *ATAs* (see subsection 5.3.3). Yet, EAs are usually applied on genomes formalized using specific encoding (binary, real numbers).

Secondly, the initial population of an EA is created by choosing randomly a set of genomes. In our case, genomes are composite transformations CTs, where each CT is composed of a set of atomic transformation alternatives $ATAs$. In addition, each ATA is composed of a set of transformation rule instantiations $TRIs$ (see definition 5) that must fulfil a set of structural constraints (see section 5.2.2). Creating randomly a population of composite transformations while checking structural constraints satisfaction may be time consuming. In case a large number of possibilities exists, but only a few of them are **valid**, i.e., satisfy the structural constraints, a random creation may not converge.

Thirdly, when applying genetic operators on composite transformations another difficulty is to ensure that the resulting composite transformations are valid. The crossover operator aims to create new solutions from existing ones by (i) choosing randomly a cutting point in the existing solutions; and (ii) exchanging the parts of the solutions beyond this point. Without considering structural constraints, crossover can be done by cutting genes at any point. As a consequence, crossover points are traditionally chosen randomly. However, when constraints exist among the values of genes, the genomes resulting from a crossover may violate these constraints, and the probability to obtain a good crossover point may be low.

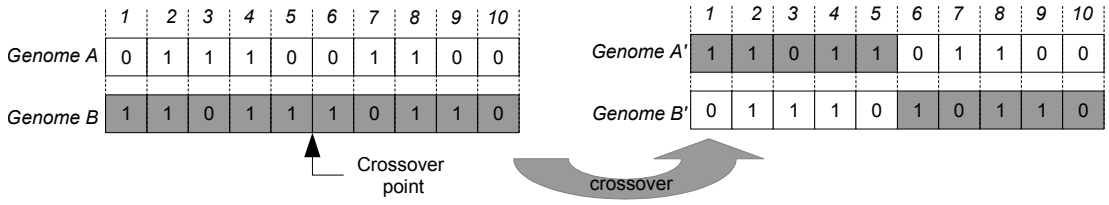

Figure 6.1: Crossover operator applied on a binary genome

A single-point crossover operation is illustrated on figure 6.1: two binary genomes, A and B, are cut between positions 5 and 6 to produce two new genomes A' and B'. In addition, consider the following constraint: values in positions 1, 5, and 6 must be equal. Genomes A and B respect this constraint, but it is not the case in genomes A' and B' resulting from the crossover.

The same problem exists with the mutation operator where we have to mutate the $TRAs$ of a composite transformations and check the *validity* of the resulting composite transformations.

To overcome these issues, we define an encoding of CTs that guarantees the respect of structural constraints when (i) creating randomly the initial population, and (ii) applying randomly genetic operators. We explain this structure in the subsection 6.2.1, and the application of selection and genetic operators in subsections 6.2.2 and 6.2.3.

6.2.1 Encoding of composite transformations

To apply EAs principles (population creation, selection and genetic operators) on a set of composite transformations we adopt the following encoding. We group $TRIs$ involved in the same structural constraints into **partitions**, and extract, from each partition, a **pool** of interchangeable sets of confluent $TRIs$. The elements of these pools are **Atomic Transformation Alternatives** ($ATAs$). We create a CT by choosing randomly one ATA in each pool of $ATAs$. To summarize, a genome (a CT in our case) becomes an ordered set of genes ($ATAs$). $CT = \{ATA_i\}_{i=1..n}$, where ATA_i is an identifier of an element in the i^{th} pool of interchangeable $ATAs$. Using this representation, and considering these $ATAs$ as genes, we guarantee that the application of genetic operators preserves the *validity* of composite transformations.

As stated in subsection 5.3.3, the creation of $ATAs$ relies on equation 5.9. First, we reorganize this equation under a conjunctive normal form: a conjunction of boolean expressions, each expression being a disjunction of boolean variables (optionally completed

with the negation operator). In our case, boolean variables are of type $Select(TRI_i)$. We call B the set of boolean expression in the conjunction, and build a partition of B: we group such expressions into smallest non-empty subsets of B in such a way that every TRI is used in expressions of one and only one of the subsets. These subsets are called the parts of the partition.

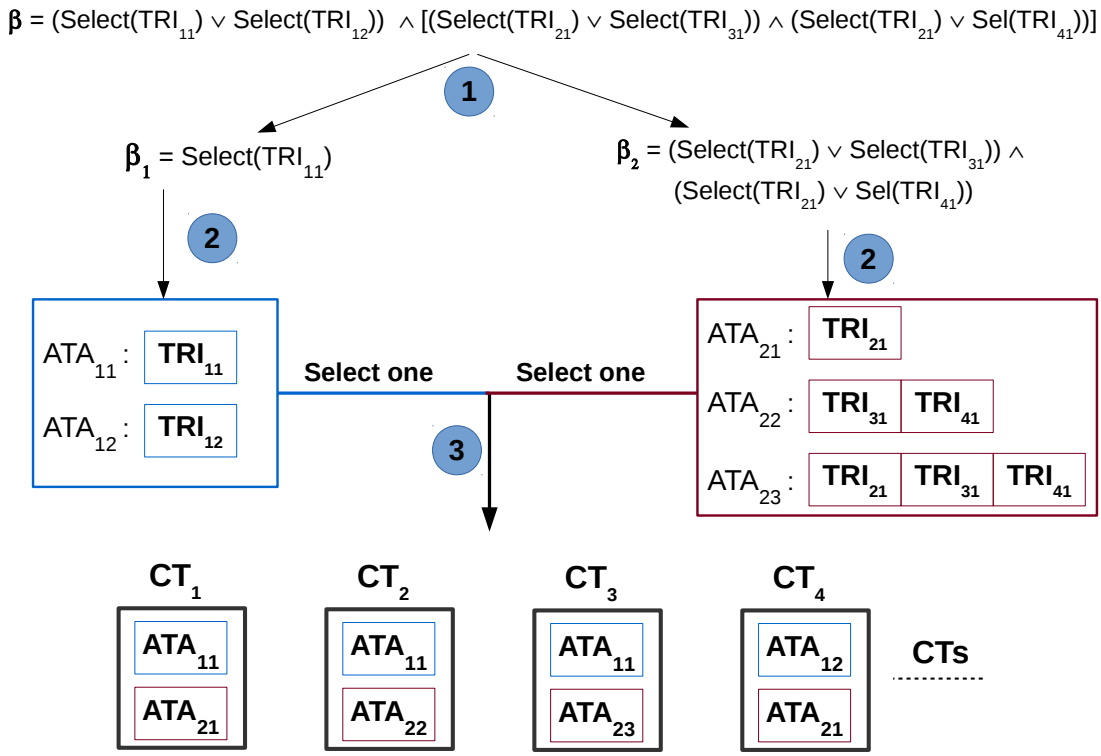

Figure 6.2: Encoding of composite transformations

To better understand this encoding, we illustrate it in figure 6.2. At the top of the figure 6.2, we have the following boolean expression $\beta = [(Select(TRI_{11}) \vee Select(TRI_{12})) \wedge [(Select(TRI_{21}) \vee Select(TRI_{31})) \wedge Select(TRI_{21}) \vee Select(TRI_{41})]$. Partitioning the boolean expression β (see bullet 1 in figure 6.2) leads to two parts: $\{Select(TRI_{11}) \vee Select(TRI_{12})\}$ and $\{Select(TRI_{21}) \vee Select(TRI_{31}), Select(TRI_{21}) \vee Select(TRI_{41})\}$. For each part of the partition, we rebuild a boolean formula β under a conjunctive normal form: expressions of the conjunction are elements of the part (see bullet 1 in figure 6.2): $\beta_1 = Select(TRI_{11}) \vee Select(TRI_{12})$ and $\beta_2 = (Select(TRI_{21}) \vee Select(TRI_{31}) \wedge Select(TRI_{21}) \vee Select(TRI_{41}))$.

A pool of atomic transformation instantiations is finally produced automatically by instantiating, for an equation β_q, all the combinations of boolean values for $Select(TRI_i)$

that satisfy β_q (see bullet 2 in figure 6.2). The solution we propose to achieve that is to use a SAT solving technique: SAT4j. Applying SAT4j for each equation β_q, we produce a set of solutions satisfying the boolean formula defined in Equation 5.9, and each solution represents an *ATA*.

For instance, applying SAT4J to β_1 leads to two *ATAs*: $ATA_{11} = \{TRI_{11}\}$, $ATA_{12} = \{TRI_{12}\}$, and applying SAT4J to β_2 leads to three *ATAs*: $ATA_{21} = \{TRI_{21}\}$, $ATA_{22} = \{TRI_{31}, TRI_{41}\}$, $ATA_{23} = \{TRI_{21}, TRI_{31}, TRI_{41}\}$. We then organize *CTs* by choosing, for each equation β_i, one *ATA* in the corresponding (i^{th}) pool (see bullet 3 in figure 6.2): $CT_1 = \{ATA_{11}, ATA_{21}\}$, $CT_2 = \{ATA_{11}, ATA_{22}\}$, etc.

Using the definitions above, we can easily apply EAs steps on a design space of *CTs* by (i) representing *CTs* as genomes, where each genome is composed of an ordered sequence of genes: *ATAs*; (ii) selecting *CTs* according to their *NFPs* evaluation; and (iii) creating new genomes by mixing their genes (crossover) and/or randomly changing their genes (mutation) while preserving validity of composite transformations. In the next paragraphs, we present the selection method and genetic operators we used to produce new *CTs*.

6.2.2 Selection of composite transformations

The selection operator is a mechanism whereby two genomes are chosen randomly from the population and the best genome regarding the objective functions (i.e., *NFPs*) is selected as parent. If the goal of the optimization problem is to minimize the results of the objective functions, the genomes (composite transformations *CTs*) having the smallest results regarding all the objective functions are the winners of the selection.

In case no genome is better than the others regarding the objective functions, we use another genetic property called **crowding distance** [39] in order to find the best genomes. The crowding distance of a genome g_i measures the distance between g_i and its neighbours (i.e., closest genomes). Large crowding distances improve the diversity of the population. More formally, the crowding distance is calculated as follows. Initially, we have a population G of N genomes, and M objective functions (F). For each objective function $F_j, j = \{1, .., M\}$, the genomes are first sorted from the lowest to the highest based on their value for the objective function F_j. The first genome and the last genome in the rank are assigned the crowding distance = infinity (i.e., boundary genomes have the best crowding distance are always selected). Then, for each remaining genome, the crowding distance is computed by (i) performing a normalized difference between the successor and predecessor genomes for each objective function F_j, and (ii) summing the obtained results to the genome crowding distance:

$$d_i = \sum_{j=1}^{M} F_j(g_{i+1}) - F_j(g_{i-1}). \tag{6.1}$$

Once we obtain the crowding distance for each genome, we apply the selection tournament between each two genomes and select those having the greater overall crowding distance.

Applying the selection tournament on composite transformations is quite easy: genomes are composite transformations, and the objective functions represent the analysis of composite transformations on a set of *NFPs*.

After the selection tournament, selected parents (best genomes) are used to create new genomes using genetic operators. We present these operators in the following subsections.

6.2.3 Application of genetic operators on composite transformations

A genetic operator is an operator used in evolutionary algorithms to guide the algorithm towards a solution to a given problem. There are two main types of operators (crossover and mutation), that must work in conjunction with one another in order for the algorithm to be successful. In the following, we describe how we apply these genetic operators on a population of composite transformations.

6.2.3.1 Crossover operator

The crossover consists in producing new genomes (called off-springs) by mixing the genes of existing genomes (called parents). Crossovers are performed from two genomes, selected after the selection step. Two off-spring solutions are then created, by exchanging parts of the parent genomes: a cut point in the parent genomes is randomly determined, and all the genes beyond that point are swapped between the two parents.

Applying the crossover on a population of composite transformations, we first select two composite transformations *CTs*. Each *CT* is organized as a set of atomic transformation alternative *ATA*. From the selected *CTs*, we create two off-spring *CTs*, by exchanging their *ATAs*: a cut point in the *CTs* is randomly determined, and all the *ATAs* beyond that point are swapped between the two *CTs*. Notice also that wherever we apply the cut in a composite transformation, we always have a new *CT* that checks the structural constraints since we exchange sequences of *ATAs*.

Figure 6.3 illustrates the application of the crossover operator on two genomes CT_1 and CT_2 respectively composed of the following genes: $\{ATA_{11}, ATA_{21}, ATA_{32}\}$ and $\{ATA_{12},$

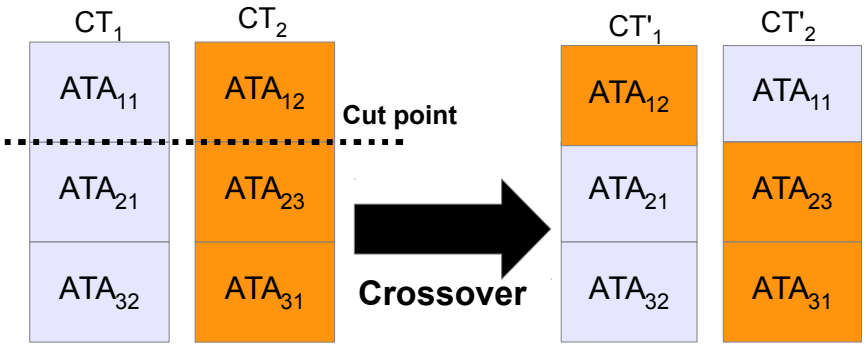

Figure 6.3: Crossover operator applied on composite transformations

$ATA_{23}, ATA_{31}\}$ to produce two new genomes CT_1' and CT_2', respectively composed of sequences: $\{ATA_{12}, ATA_{21}, ATA_{32}\}$ and $\{ATA_{11}, ATA_{23}, ATA_{31}\}$.

6.2.3.2 Mutation operator

After performing the crossover operator, the obtained genomes is mutated. The mutation operator selects a gene (randomly), and changes its values.

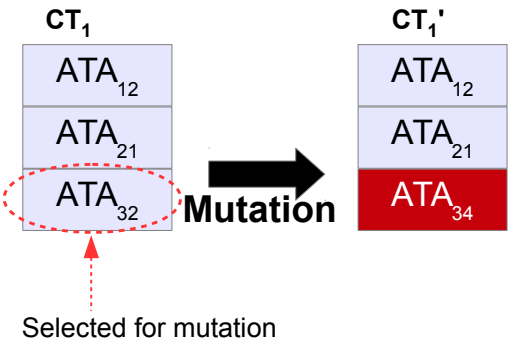

Figure 6.4: Mutation operator applied on a composite transformation

In figure 6.4, the mutation operator acts on a single genome CT_1 to obtain a mutated genome CT_1'. The third gene ATA_{32} of CT_1 is selected at random and is mutated into a new gene, ATA_{34}. ATA_{34} is basically another alternative than ATA_{32}, both being stored in the same pool of ATAs, obtained after applying SAT4J on a characteristic boolean equation β_3.

6.2.4 Identification of non-dominated composite transformations

In the paragraphs above, we defined an encoding for composite transformations in order to apply genetic operators and from that produce new valid composite transformations. Now, we apply an evolutionary algorithm in order to find a set of non-dominated composite transformation. In this thesis, we use a well-known evolutionary algorithm called NSGA-II: an improved version of NSGA [65] (Non-dominated Sorting Genetic Algorithm) to find the non-dominated composite transformations.

6.2.4.1 Non Dominated Sorting Genetic Algorithm (NSGA-II)

NSGA-II [39] is an improved version of the Non Dominated Sorting Genetic Algorithm (NSGA) [65]. More precisely, NSGA-II provides two improvements for the research of Pareto-optimal solutions: NSGA-II sorts the combination of parents and children populations with an elitist strategy (fast non-dominating sorting procedure [39]) and introduces the crowding distance to improve diversity of populations. Thanks to these improvements NSGA-II is one of the most used evolutionary algorithm. To support this conclusion, Deb et al. [39] performed several experiments using different EAs (NSGA-II, PAES, and SPEA). Based on the experimental results, the authors stated that NSGA-II is better than the other studied EAs in terms of finding diverse populations.

Generally, NSGA-II can be roughly detailed as following steps:

1. Initialization of the population based on the optimization problem.
2. A sorting process based on the evaluation of the initial population regarding the objective functions.
3. Once the sorting is complete, a crowding distance value is computed (using the equation 6.1) for each genome of the population. The genomes in population are selected based on rank and crowding distance.
4. The selection of genomes is carried out using a tournament selection. In case genomes are equivalent, the genome having the greater crowding distance to the current population is selected.
5. The creation of new genomes using crossover and mutation operators. The population of new created genomes is called **offspring population**.
6. Offspring population and current population are combined, and the solutions of the next generation are defined using Pareto-fronts. The new generation is filled by each

genome of the best fronts until the population size exceeds the current population size.

7. Repeat the six steps above until a convergence criteria is met.

In the following subsection, we describe the application of NSGA-II on a search space of composite transformations.

6.2.4.2 Application of NSGA-II on composite transformations

Applying NSGA-II steps (defined in 6.2.4.1) on CTs, we have the following steps:

1) We create randomly an initial population P_t composed of N (specified by end-users of our method) composite transformations CTs. We evaluate each CT regarding $NFPs$ by executing it and evaluating the corresponding target model.

2) We sort P_t based on a non-domination criterion, using evaluated $NFPs$.

3) We create an off-spring population Q_t of size N (the same size as P_t) using selection tournaments, crossovers and mutations. We apply each CT of Q_t to the input model and evaluate the $NFPs$ of the corresponding target models.

4) We create a new population R_t of size 2*N by gathering elements from P_t and Q_t ($R_t = P_t \cup Q_t$).

5) We sort R_t with a fast non-dominated sorting procedure [39], to identify non-dominated fronts $\{F_i\}_{i=1..N}$ (F_1 being the best and F_N the worst front).

6) We generate the new parent population P_{t+1} of size N. The genomes belonging to the best non-dominated front, (i.e., F_1) represent the best genomes in R_t and must be emphasized more than any other genome. If the size of F_1 is smaller than N, all solutions of F_1 are inserted in P_{t+1}. Then, the remaining population of P_{t+1} is chosen from subsequent non-dominated fronts in order of their ranking: the genomes of F_2 are chosen next. In some cases, not all the genomes from a front F_i can be inserted in population P_{t+1}: the number of empty slots of P_{t+1} is smaller than the number of genomes in F_i. In order to choose which ones are selected, these genomes are sorted according to their crowding distance [39], and the genomes having the highest crowding distance are used to fill the empty slots of P_{t+1}.

7) The created population P_{t+1} is then used to create a new population Q_{t+1}. We repeat the process (1 to 7) to the next generation until a convergence criterion is met, and a set of non-dominated composite transformations is identified.

In this section, we presented an automatic approach to identify non-dominated composite transformations using $NSGA-II$. However, this approach is limited to the applica-

tion of one model transformation only (e.g., the allocation of software components), while model transformations are often structured as chains. We present in the next section the application of our approach on model transformation chains.

6.3 Exploration of chained model transformations

In order to ease their maintenance, improve their reusability and scalability, model transformations are often structured as model transformation chains [15]. A model transformation chain is specified as linked model transformations that generate the target model. For instance, consider a model transformation composed of L **links**. The source model is transformed into $L-1$ intermediate models by executing $L-1$ links: each link generates an intermediate model on which we apply another link, and we repeat the process until we reach the intermediate model generated by the link $L-1$. Finally, the execution of the last link L on the $(L-1)^{th}$ intermediate model generates the target model with the lowest abstraction level.

Optimizing a chain of model transformations cannot be reduced to apply the optimisation method (performed in the previous section 6.2) on each link of the model transformation chain. This is mainly due the following sub-problems:

1. An important step in EAs is the analysis of the genomes regarding objective functions. In our case genomes are model transformations of the chain and the objective functions are NFPs. Since in a model transformation chain we have several intermediate models, we can evaluate each intermediate model regarding a specific NFP. However analysing the intermediate model generated by a link of the chain may be incorrect in two situations.

   Firstly, let assume that the intermediate model generated by a link is analysable regarding a set of NFPs. However, other links may impact these NFPs. In this case, the analysis made on intermediate models is incorrect. Thus, we cannot analyze the intermediate model of a link without executing other links since links of a model transformation chain are interdependent.

   Secondly, let assume that the intermediate model generated by a link of the chain cannot be analyzed because it does not contain all the necessary information for analysis. In this case models resulting from the model transformation chain are only analyzable if all the links of the chain are executed. Thus, evaluation of NFPs on intermediate models is impossible.

2. Optimizing a model transformation chain, each link produces an unknown number of architectural elements, and therefore an unknown number of transformation rule instantiations *TRIs* (see section 6.2) for the next link. Thus, we cannot ensure that applying the genetic operators (presented in subsection 6.2.3) on composite transformations of a model transformation chain leads to new valid composite transformations.

The solution we propose to solve these two sub-problems is to optimize (by applying the EA) each link L_i of a model transformation chain $(L_1, L_2, .., L_N)$ using the optimization results of the next links of the chain: L_{i+1} .. L_N. The application of the EA on a link L_i is configured with the following parameters:

- $\mathbf{pop_{L_i}}$: the size of the populations for link Li.
- $\mathbf{max_{L_i}}$: the maximum number of iterations of the EA applied on link L_i. It represents a convergence criterion that limits the number of iterations of the EA in case it does not converge.
- $\mathbf{max\_stable_{L_i}}$: the maximum number of iterations of the EA applied on link L_i without evolution of the population. It represents a convergence criterion.
- ($\mathbf{pop_{L_i} < max_{L_i}}$) and ($\mathbf{max\_stable_{L_i} < max_{L_i}}$).

6.3.1 Exploration of links

In order to explain the exploration of a model transformation chain composed of N links, we use an inductive reasoning. More precisely, we define our solution to explore (i) the last link L_N, and (ii) the link before last (L_{N-1}). Using the exploration results we demonstrate that we can explore all the links of the chain.

6.3.1.1 Exploration of the last link of a chain

We first apply the EA on the last link of the chain: $\mathbf{L_N}$. The application of the EA in this case was presented in section 6.2. It consists in executing the transformation link L_N on a model. The execution of the link L_N produces a set of composite transformations. These composite transformations are then explored using the steps defined in subsection 6.2.4.2: a parent population of composite transformations having a size pop_{L_N} is created. Composite transformations are then evaluated regarding the NFPs by executing them, and analysing the corresponding target models. The evaluated composite transformations are then used to create an offspring population. Parent and offspring populations

are combined and the non-dominated composite transformations are selected. This process is repeated until the EA converges (max_{L_k} or $max\_stable_{L_k}$), and a population (of size pop_{L_N}) of non-dominated composite transformations is created.

To summarize, when applying the EA on the last link of the chain L_N, we identify pop_{L_N} composite transformations. These composite transformations can be analysed regarding the NFPs of interest by generating the corresponding target models, and analysing these models.

6.3.1.2 Exploration of the link before last

After explaining how to explore the last link of a model transformation chain, we apply the EA on the link before last L_{N-1}: $\mathbf{L_k = L_{N-1}}$. Applying the EA for this link L_k, we produce at most max_{L_k} composite transformations. These composite transformations must be evaluated regarding the NFPs in order to determine the non-dominated ones. Unlike for the link L_N (see subsection 6.3.1.1), we must apply the following link L_{k+1} in order to evaluate the composite transformations of the link L_k.

More precisely, each composite transformation $CT^i_{L_k}$ (with $i \in [1, max_{L_k}]$) of the link L_k is executed and generates an intermediate model. This model is enriched by executing the link L_{k+1} (the last link L_N) on it. Applying the EA for the link L_{k+1} (see subsection 6.3.1.1), we identify $pop_{L_{k+1}}$ non-dominated composite transformations ($CT^j_{L_{k+1}}$ with $j \in [1, pop_{L_{k+1}}]$). Once executed these composite transformations generate $pop_{L_{k+1}}$ analysable target models. Thus, we apply **pairs of composite transformations** ($\{CT^i_{L_k}, CT^j_{L_{k+1}}\}$) in order to generate analysable target models, and thus evaluate the composite transformations of the link L_k.

Since we evaluate at most max_{L_k} composite transformations for the link L_k, we have to identify at most ($max_{L_k} \times pop_{L_{k+1}}$) pairs of composite transformations. These pairs of composite transformations are compared together, and the best pop_{L_k} pairs of composite transformations are selected and executed to generate the optimal (or near optimal) target models.

6.3.2 Application example

In order to better understand the exploration of an intermediate transformation link, we consider a model transformation chain composed of two links L_1 and L_2. Note that, each link of the chain has its own parameters for the application of the EA. These parameters are defined in the top of the figure 6.5.

6.3.2.1 Creation of parent population of the link L_1

Initially, we create the parent population for the link L_1 (see on the left part of the figure 6.5). This population has a fixed size of 3 ($pop_{L_1} = 3$). In our example the parent population is composed of the following composite transformations: $CT^1_{L_1}$, $CT^2_{L_1}$, and $CT^3_{L_1}$. From this parent population, we create the offspring population. To perform that, we need to execute each composite transformation, generate the corresponding target model and analyse the generated model regarding the NFPs. Yet, the generated models do not contain enough information to be evaluated regarding the NFPs. To solve this problem, we enrich the target models generated in the link L_1 by applying the link L_2 on them. Then, we apply the EA on the link L_2, and identify $pop_{L_2} = 2$ non-dominated composite transformations. The execution of these composite transformations generate pop_{L_2} non-dominated target models. More precisely, these target models were obtained by applying pairs of composite transformations: $\{CT^1_{L_1}, CT^1_{L_2}\}$, $\{CT^1_{L_1}, CT^2_{L_2}\}$,..,$\{CT^3_{L_1}, CT^{18}_{L_2}\}$. These pairs represent the association of one composite transformation of the link L_1 with $pop_{L_2} = 2$ non-dominated composite transformations of the link L_2. These pairs of composite transformations are then used to create the offspring population of the link L_1.

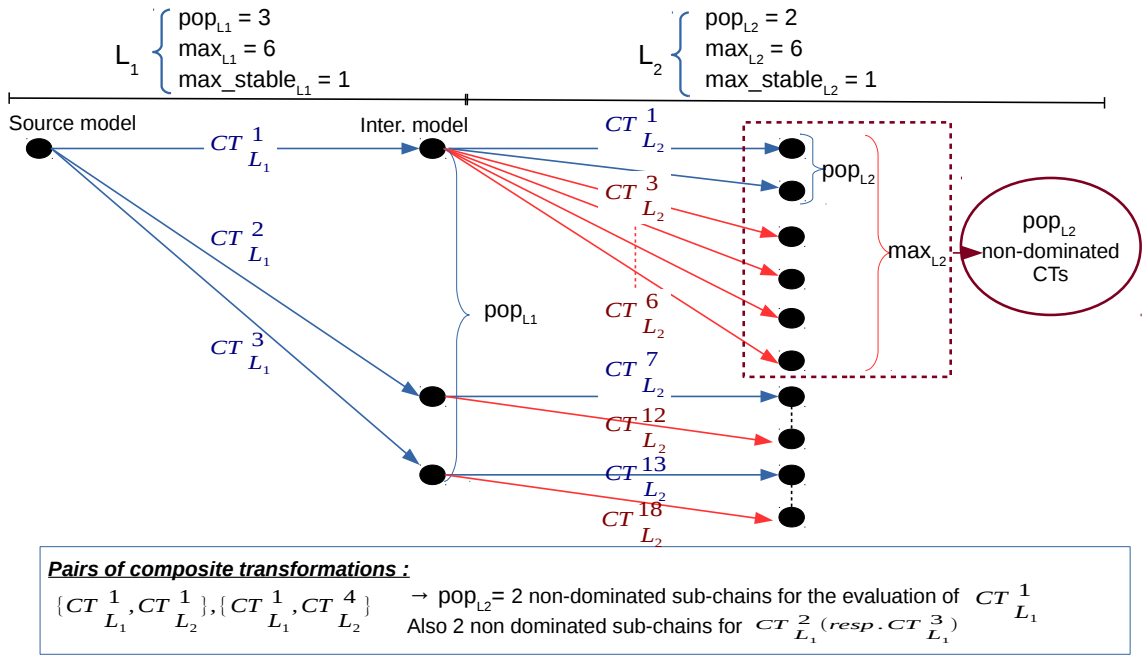

Figure 6.5: Creation and evaluation of parent population for the link L_1

6.3.2.2 Creation of offspring population of the link L_1

The identified pair of composite transformations are compared with each other, and the composite transformation of the link L_1 contained in the best pairs are selected to create the offspring composite transformations. In our example, $\{\mathbf{CT^1_{L_1}}, CT^4_{L_2}\}$, $\{\mathbf{CT^1_{L_1}}, CT^6_{L_2}\}$, and $\{\mathbf{CT^2_{L_1}}, CT^8_{L_2}\}$ are selected to create the offspring composite transformation $CT^4_{L_1}$, CT^5_{L1} and $CT^6_{L_1}$ (see on the left part of the figure 6.6). Each offspring composite transformation is then evaluated regarding the NFPs by identifying pairs of composite transformations.

We repeat the creation of parent and offspring composite transformations until reaching the convergence criteria of the link L_1: $max_{L_1} = 6$. In order to evaluate the composite transformations of the link L_1 we produced $(max_{L_1} \times pop_{L_2}) = 12$ pairs of composite transformations. These pairs of composite transformations are compared, and the best pop_{L_1} pairs are selected to generate the optimal (or near optimal) target models (or architecture alternatives).

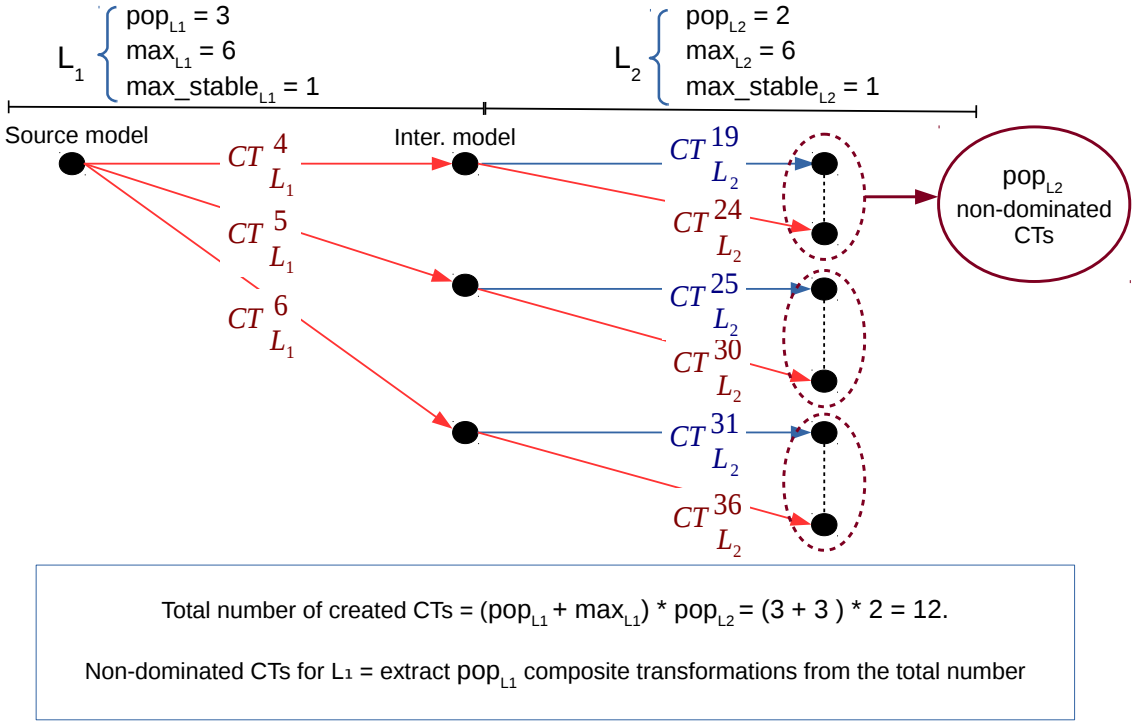

Figure 6.6: Creation of offspring population for the link L_1

6.3.3 General formalization

In order to show the applicability of our exploration approach for model transformations chains composed of N links (with $N > 2$) we use an inductive reasoning.

In section 6.2, we described our solution to explore a model transformation chain composed of one link ($N = 1$). We assume that we can explore a model transformation chain composed of N links: L_1 .. L_N. This represents our inductive hypothesis. We now consider a model transformation chain composed of $N+1$ links: L_1 .. L_{N+1}. Using the induction hypothesis, we can explore a model transformation chain composed of N links: L_2 .. L_{N+1}. The exploration of this chain works as follows.

We first explore the link L_{N+1}. We produce a set of composite transformations for this link. Then, we apply the EA on these composite transformations, and identify $pop_{L_{N+1}}$ non-dominated composite transformations. After exploring the link L_{N+1}, we explore the link L_N. The composite transformations produced by the link L_N cannot be evaluated regarding the NFPs. To evaluate such composite transformations, we identify pop_{L_N} pairs of non-dominated composite transformations (as defined in subsection 6.3.1.2). We use the same steps until reaching the link L_2. The difference here is that we identify sub-chains of composite transformations in order to evaluate this link. These sub-chains have the following structure: {the application of a composite transformation of the link L_2, the application of a composite transformation of the link L_3, ..., the application of a non-dominated composite transformation of the link L_N, the application of a non-dominated composite transformation of the link L_{N+1}}. These sub-chains are then compared, and the best pop_{L_2} sub-chains of composite transformations are executed and generate a set of non-dominated target models.

Using the results of the exploration of the chain L_2,..,L_{N+1}, we can explore the link L_1, and thus explore a model transformation chain composed of $N+1$ links. More precisely, for each composite transformation $CT^i_{L_1}$ (with $i \in [1, max_{L_1}]$) of the link L_1, we identify pop_{L_2} sub-chains of composite transformations. In total, we produce at most $(max_{L_1} \times pop_{L_2})$ sub-chains of composite transformations. These sub-chains of composite transformations are then compared, and the pop_{L_1} best sub-chains of composite transformations are selected and executed to generate optimal (or near optimal) target models.

6.4 Conclusion

In this chapter we have detailed the second and third contributions of our work: the exploration of composite and chained model transformations using EAs.

For the second contribution, we explore a search space of composite transformations, and select only those optimizing a set of conflicting non functional properties. Optimizing manually composite transformations can be tedious and error-prone. To overcome this difficulty, we use evolutionary algorithms for design space exploration. More precisely,

we apply EAs on a design space of composite transformations in order to identify the non-dominated composite transformations. Once executed, the non-dominated composite transformations generate a set of optimal architecture alternatives.

The main steps in the application of EAs on composite transformations are (i) the encoding of composite transformations, (ii) the application of the selection and genetic operators on composite transformations, and (iii) the identification of the non-dominated composite transformations using a well-known EA called NSGA-II.

For the first step, we relied on the equation 5.9 and the solver SAT4j, in order to (i) group *TRIs* involved in the same structural constraints into partitions, and (ii) extract from each partition, a pool of interchangeable sets of confluent *TRIs* (also called atomic transformation alternatives). Then, the produced atomic transformation alternatives are composed to produce valid composite transformations. Once composite transformations are well-encoded, we apply the selection and genetic operators on them in order to produce new valid composite transformations. Finally, we use the steps of a well-known EA called NSGA-II to identify the non-dominated composite transformations.

For the third contribution, we structure model transformations as chains and explore these chains in order to identify the non-dominated ones. To perform this exploration, we apply EAs steps (selection and genetic operators) over links of a model transformation chain, and produce the most suitable composite transformations for each link. The set of non-dominated composite transformations produced for the first link of the chain generates the set of optimal (or near optimal) architecture alternatives.

This concludes the presentation of the technical contributions of this thesis which were detailed throughout chapters 5 and 6. We now move to the experimental evaluation of our proposals in the next chapter 7.

7 Experiments and Evaluation of the approach

CONTENTS

7.1 Introduction

Having detailed our contributions in the previous chapters, we now present the validation activities that we carried out to assess our proposals. In order to evaluate and validate our approach, we propose to (i) formulate a set of research questions aiming at evaluating our approach regarding some criteria, (ii) apply our approach on experimental case studies, and (iii) answer to the formulated research questions using the experimental results. Thus, we first enumerate in section 7.2 the research questions we propose to answer. Then, we define different case studies, apply our approach on these case studies, and evaluate (i) the genericity of our framework in section 7.3, and (ii) the quality of our framework in section 7.4.

7.2 Research questions

As stated in chapter 3, we develop a framework that explores a design space of architecture alternatives by composing model transformations, validating a-priori composite transformations and applying evolutionary algorithms on valid composite transformations. In order to evaluate the quality of the developed framework, we introduced the following research questions (RQs):

RQ1: can our approach be reused to solve different types of optimization problems? This research question aims at evaluating if the proposed framework is generic. In other words, can we implement different rule-based model transformations, and apply our framework on these model transformations to identify optimal (or near optimal) architecture alternatives.

RQ2: how efficient is it to proceed to structural constraints validation a priori instead of a posteriori? This research question aims at evaluating whether using EAs to explore a design space made up of architectural candidates is more efficient to validate **a priori** rather than **a posteriori**.

RQ3: what is the distance between results obtained with our approach and other approaches such as linear programming or random exploration algorithms? This research question aims at evaluating the intrinsic quality of architectural alternatives found with our framework.

RQ4: how much resource (memory and computation time) does our approach require to explore a large design space of composite transformations? This research question aims at evaluating the complexity of our approach, in terms of computation time and memory space.

In order to answer to these questions we propose to first define experimental case studies. Then, we apply our framework on these case studies. Finally based on the experimental results, we evaluate our framework. The evaluation of our framework is described in the following sections.

7.3 Evaluation of the genericity of the framework

In order to answer to the first research question (see RQ1 in section 7.2), i.e., the genericity of our framework, we define two different case studies (having different optimization problems), apply our framework on these case studies, and check whether our framework identifies optimal (or near optimal) architecture alternatives for both case studies.

Applying our framework on these case studies, we demonstrated the capability of our approach to explore a design space of composite transformations obtained from the implementation of different rule-based model transformations.

In the remaining of the section, we detail the architecture model, the applied rule-based model transformations, and the results of the application of our approach for both case studies.

7.3.1 Model refinement case study

In the first case study, railway designers want to reduce the time interval separating two consecutive trains while guaranteeing passengers safety. To reach this objective, the adaptation of multi-core architecture gives the opportunity to embed more computation power on-board trains. Functions traditionally deployed on the wayside infrastructure can then be migrated on-board in order to reduce response time of functions. However, this migration of functions raises new issues: maintenance of software embedded on trains is more difficult and costly than for equipments on the wayside. In addition, grouping functions on-board the train should not lead to hardware resources overconsumption otherwise the safety of the system may be put at risk. To facilitates the migration of functions, and their interactions while improving a set of NFPs, we represent them using models. More precisely, we propose to rely on model refinement transformation for source code generation. This type of model transformation allows to (re)-evaluate the NFPs of the case study: software maintenance cost and resources (memory and execution time) consumption.

7.3.1.1 System model

The architecture of the considered real-time embedded system in the first case study consists of the software architecture of a train control system. This software architecture is made up of two interconnected processes: the Automatic_Train_Operation (ATO) process, responsible for controlling the position, speed, and acceleration of the train. The other process, called Automatic_Train_Protection (ATP), communicates with the ATO in order to check the validity of data computed by the ATO. Figure illustrates the AADL architecture of the Automatic_Train_Operation (ATO) process. This process is made up of four interconnected (by AADL ports) AADL threads aiming at controlling the position, speed, and acceleration of the train. In addition ATO and ATP are connected by means of AADL connections between their ports (see dangling ports on the right part of figure 9.5).

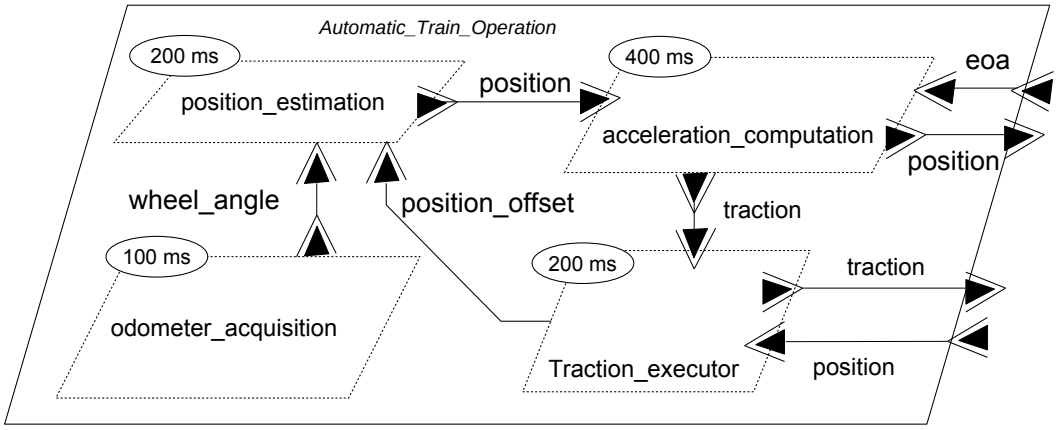

Figure 7.1: System model of the first case-study

7.3.1.2 Model transformations definition

On the architecture model presented in subsection 7.3.1.1, we apply an AADL to AADL model transformation that implements a model refinement for code generation. This transformation is made up of 120 ATL matched rules, and provides three variants: LowET (Low Execution Time), LowMF (Low Memory Footprint), and LowMC (Low Maintenance Cost) that represent the most efficient solutions in terms of execution time, memory footprint and maintenance cost respectively. Each transformation alternative transforms AADL ports and connections between thread subcomponents of the same process.

Both **LowET** and **LowMF** use the same mathematical function which implements communications among threads. This mathematical function computes the position in the queue where messages have to be stored using pre-computed slot indexes. However,

LowET is based on static lookup tables, and **LowMF** is based on runtime services. More precisely, the **LowET** version implements communications among threads using static lookup tables. These tables store, for each activation of a thread, the position of the data queue where a message should be stored or retrieved. Data is stored in the queue with pre-computed slot index [66]. *"Slot indexes are computed with formulas attaching to each message a unique sequence number. These formulas allow computing indexes concurrently on each thread without synchronization between them"*. Using pre-defined slots allow to implement a deterministic communication protocol, that is very important to assess safety in critical embedded software applications (such as an automated train control). This implementation is very efficient in terms of execution time. Yet, this solution is memory consuming: lookup tables have to be stored in memory. In addition, it is quite difficult to maintain since the lookup tables depend from the task-set. The **LowMF** version implements communications among threads using runtime services, which is less memory consuming. Yet, this version is also less maintainable since for each new task-set the runtime services have to be modified.

These two solutions (*LowEt* and *LowMF*) are compatible since they use the same mathematical function to implement threads communications. However, maintaining these solutions requires a very good understanding of the communication protocol. They are thus more complicated than solutions based on classical runtime services.

The **LowMC** version is based on a linked list to implement the shared queue, and uses operating system semaphores to protect accesses to this queue. This solution does not depend on the task-set. Thus, it is more easier to maintain. Yet, it is less efficient in terms of execution time and memory footprint because of the use of semaphores.

Because of the principles of these three transformation alternatives, the LowMF alternative is easily composable with the LowET (and vice versa) but their composition with the LowMC alternative is more delicate. As a consequence, we had to express structural constraints among these transformation alternatives in order to produce consistent composite transformations. They were expressed using the TRC (defined in the previous chapter). This TRC describes the following structural constraints:

1. if an output port p is transformed by a rule from *LowMF* or *LowET*, then all the ports p is connected to must be transformed by a rule from *LowMF* or *LowET*. This structural constraint expresses the compatibility of *LowMF* with *LowET* for transforming pairs of connected ports. Indeed, rules from *LowMF* and *LowET* implement communications using services and data structures presented in [67]. Two components can thus communicate correctly with one using the *LowMF* implementation, and the other using the *LowET* implementation.

2. if an output port p is transformed by a rule from *LowMC*, then all the ports p is connected to must be transformed by a rule from *LowMC* as well. This structural constraint expresses the incompatibility of *LowMC* with *LowMF* and *LowET* for transforming pairs of connected ports.

7.3.1.3 Experimental results

The experimentation begins from the architecture model and the model transformations presented in previous subsections. We suppose that the source and target models contain AADL properties to evaluate the *NFPs* of interest: CPU resource consumption, memory footprint, and maintainability. In addition, the TRC formalizes structural constraints among model transformation rules. From these input artefacts, our framework identifies transformation rule instantiations *TRIs*, detects confluent *TRIs*, and produces boolean formulas (as presented in section 5.3) to create atomic transformation alternatives *ATAs* (as presented in section 6.2.1).

Thread	Timing budget (ms)	Memory budget (Bytes)	Maintenance budget (hours)
acceleration_computation	400	150	160
position_estimation	200	150	120
traction_executor	200	150	60
odometer_acquisition	100	50	90
position_correlation	400	200	200
traction_correlation	200	600	120
protection_computation	200	30	15
logging	200	250	170

Table 7.1: Setup of the input architecture

NSGA-II is then applied in order to find non-dominated composite transformations (*CT*): (i) the initial population, composed of 10 *CTs*, is constructed by choosing *ATAs* randomly; (ii) the crossover probability is set to 0.9; (iii) the mutation probability is 0.05 ($1/nATAs$, nATAs being the number of *ATAs* in a *CT*; nATA = 19 in our example); and (vi) the maximum number of generations is set to 100 (stopping criterion). Notice that, these parameters were used in most experimental setups we found in research works evaluating optimization frameworks based on NSGA-II. They can therefore be considered as default values. The impact of model transformations on the considered NFPs is presented in terms of margins of NFPs:

$$Margin_{NFP} = \frac{Budget_{NFP} - Consumption_{NFP}}{Budget_{NFP}}.$$

Budgets of each NFP have been given in table 7.1. The consumption of each NFP is computed as follows: CPU consumption is computed by relying on schedulability tests that enables to compute the CPU usage of a set of concurrent tasks [14]. Memory consumption is the sum of the memory size of AADL data subcomponents of an AADL process component. Maintainability consumption is the sum of the maintenance cost associated with subprograms that are called in thread subcomponents of an AADL process component. We assume that this cost has been computed, for each subprogram, prior to the execution of our approach.

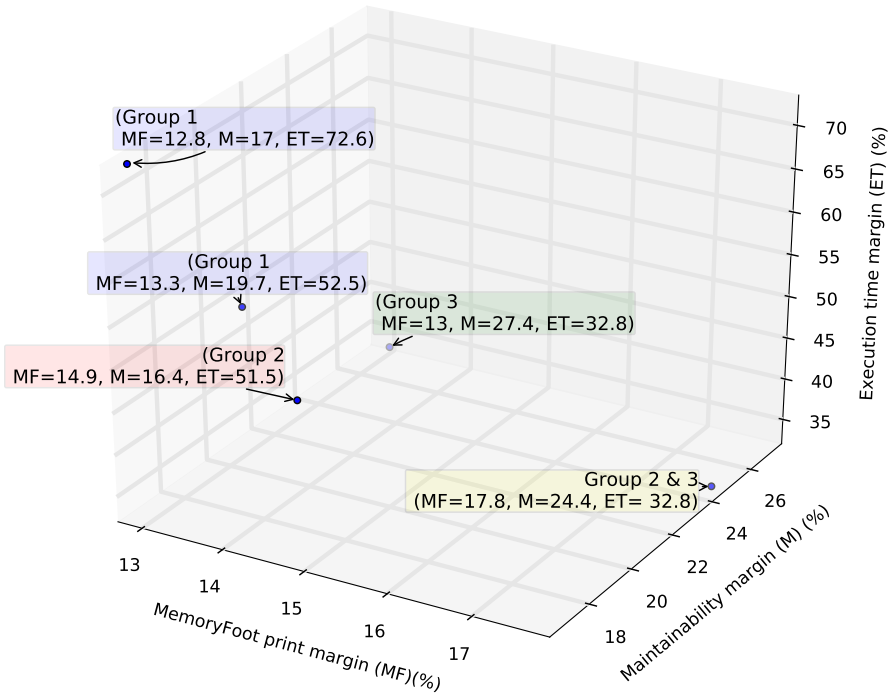

Figure 7.2: Experimental Results

Figure 9.6 shows that our approach exhibits a set of 5 solutions, each of them presenting non-dominated solutions in terms of NFPs. Indeed, in this figure we can observe that solutions from group 1 have a better margin in terms of execution time, but a lower margin in memory and maintainability. Solutions from group 2 have a better margin in memory (but a lower one in maintainability and execution time). Finally solutions from group 3

have a better margin in maintainability (but a lower on in execution time and memory). Note that some solutions belong to two groups as they have good performances for two NFPs, but lead to a worse solution with respect to the third NFP.

7.3.2 Safety case study

In the second case study, railway designers want to organize all the operational resources so that their system attains the following *NFPs*: safety, reliability, availability and response time. To perform that, designers use two different design decisions. Firstly, the application of safety design patterns which intended to replicate software components in order to increase the reliability and/or safety of the system. Secondly, the allocation of software components (replica) on hardware components: initially, software components are not allocated on any hardware components, and designers have to decide, for each software component, on which hardware component it will be allocated in order to maximize reliability and availability while minimizing response time.

7.3.2.1 System model

The architecture of the considered real-time embedded system in the second case study consists of a hardware model (physical architecture) and a software model (logical architecture).

The **hardware model** consists of a set of processors $H = \{h_1, h_2, ..., h_{N_H}\}$ connected through communication buses $B = \{\beta_1, \beta_2, ..., \beta_{N\beta}\}$. Processors run tasks scheduled with a preemptive fixed-priority scheduling, and communications on a bus are scheduled with a non-preemptive, priority based, policy. Figure 7.3 (a) gives an illustration of a simple hardware model using the AADL graphical notation. This model is made up of four processors (h_1, h_2, h_3 and h_4) and a bus (β_1) that connects these processors.

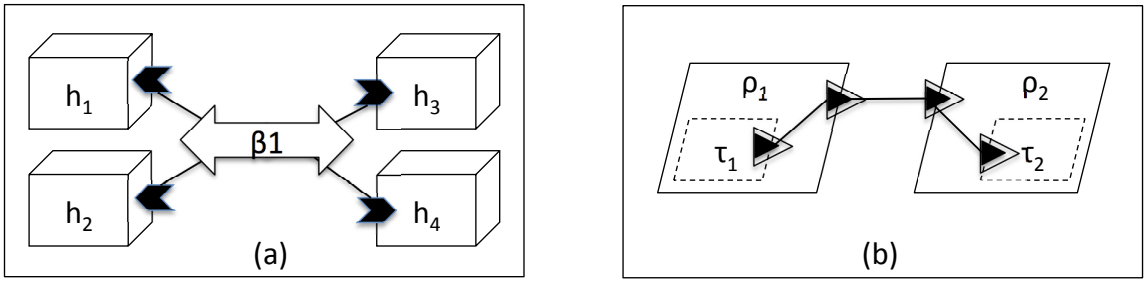

Figure 7.3: (a) hardware and (b) software AADL models

The **software model** consists of a set of process components $P = \{\rho_1, \rho_2, ..., \rho_{N\rho}\}$, and a set of threads $T = \{\tau_1, \tau_2, ..., \tau_{N\tau}\}$. Each thread τ_i is activated periodically with a period

φ_{τ_i} and is characterized by a fixed priority π_{τ_i}. Each thread τ_i has a local deadline LD_{τ_i} that represents the maximum time value allowed for the associated thread to be executed. $Proc(\tau_i)$ refers to the process to which the thread τ_i belongs.

In addition, communication between threads can be established with messages. $M = \{\mu_1, \mu_2, ..., \mu_{N\mu}\}$ is the set of messages exchanged in the software model. In turn, a message μ_i is defined by an activation period φ_{μ_i} and a fixed priority π_{μ_i}. $Src(\mu_i)$ and $Dest(\mu_i)$ return, respectively, the sender and the receiver thread of message μ_i.

Threads and messages are, respectively, characterized by a vector of worst-case execution times (WCETs) $\vec{\omega_{\tau_i}} = \{\omega_{\tau_i,h_1}, \omega_{\tau_i,h_2}, ..., \omega_{\tau_i,h_{Nc}}\}$ and worst-case transmission times (WCTTs) $\vec{\omega_{\mu_i}} = \{\omega_{\mu_i,\beta_1}, \omega_{\mu_i,\beta_2}, ..., \omega_{\mu_i,\beta_{N\beta}}\}$, where $\omega_{\tau_i,h_{Nh}}$ and $\omega_{\mu_i,\beta_{N\beta}}$ are respectively the WCET of τ_i on processor h_{Nh} and the WCTT of μ_i on bus $\beta_{N\beta}$. Communicating threads are structured in one or more paths that represent the end-to-end executions of the system.

Definition 9 *A directed path Γ_{τ_i,τ_j} from the thread τ_i to the thread τ_j is a sequence of threads and messages $\Gamma_{\tau_i,\tau_j} = [\tau_i, \mu_i, \tau_{i+1}, \mu_{i+1} ..., \mu_{j-1}, \tau_j]$.*

$\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_{N_\Gamma}\}$ denotes the set of system paths. Figure 7.3 (b) gives an illustration of a simple software model using the AADL graphical notation. This model is made up of two processes ρ_1 and ρ_2, each containing one thread, respectively τ_1 and τ_2. A message μ_1 is exchanged between these threads through AADL event data ports. This model thus exhibits a single system path $\Gamma_1 = [\tau_1, \mu_1, \tau_2]$.

Once the system models of both case studies are defined, we describe the model transformation alternatives we consider to improve *NFPs* of each system model. Notice that, for the first case study we used one rule-base model transformation; and for the second case study we used a chain of two rule-base model transformations.

7.3.2.2 Model transformations for the second case study

Model transformations used for the second case study implement classical design decision in critical embedded systems: the replication of software components, and their allocation onto hardware components.

1. Software components replication: we consider two model transformations for software components replication, namely: *Two-Out-Of-Three (2oo3)* and *Twice-Two-Out-Of-Two (2*2oo2)*.

When applying the **2oo3 transformation**, each process is replicated twice (the result is of three replicas: the process plus its two replicas), and then all these processes are

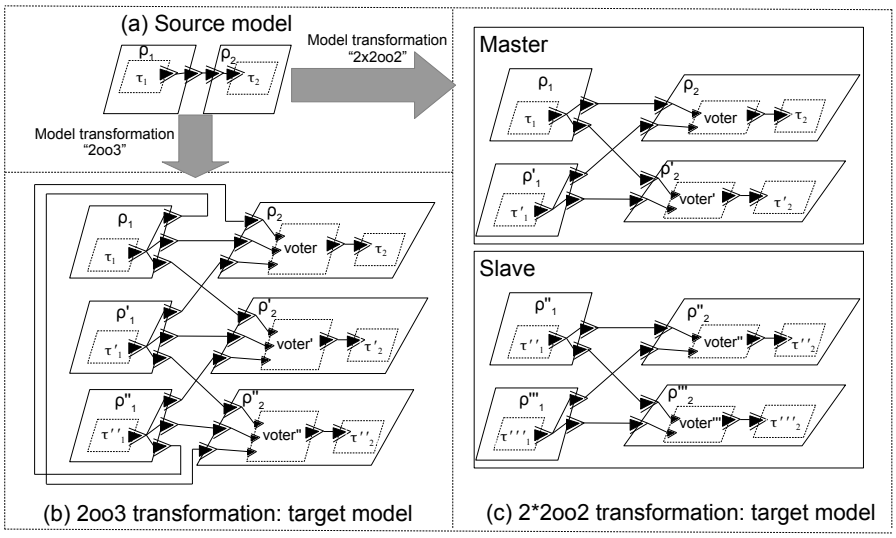

Figure 7.4: Application of replication model transformations

operating in parallel. Communications between processes are also replicated such that each process and each of its replicas communicate with their successor processes and all of their replicas.

Figure 7.4 (b) illustrates the application of the 2oo3 pattern on two communicating processes: ρ_1 and ρ_2. $\rho_1^{'}$ (respectively $\rho_2^{'}$) and $\rho_1^{''}$ (resp. $\rho_2^{''}$) are replicas of process ρ_1 (resp. ρ_2). As a result of using 2oo3 pattern, the system is considered as working if at least two of the three replicas produce the same result. To check this, a voter thread is added in ρ_2, $\rho_2^{'}$, and $\rho_2^{''}$: the voter reads the three input data received and detects/masks the occurrence of faults.

When applying the **2*2oo2 transformation**, each process is replicated three times (i.e., four replicas). Figure 7.4 (c) illustrates the application of the 2*2oo2 pattern, where $\rho_1^{'''}$ and $\rho_2^{'''}$ are respectively the third replicas of process ρ_1 and ρ_2. In 2*2oo2, process replicas are organized into two couples *Master*=(ρ_1,$\rho_1^{'}$,ρ_2,$\rho_2^{'}$) and *Slave*=($\rho_1^{''}$, $\rho_1^{'''}$,$\rho_2^{''}$, $\rho_2^{'''}$), and communications among processes of each couple. In addition, a voter thread is added in processes ρ_2, $\rho_2^{'}$, $\rho_2^{''}$, and $\rho_2^{'''}$. When the system is working, the four processes operate in parallel but one couple only produces data towards actuators (this couple is called master). The second couple operates but its data are not sent to actuators (this couple is called slave). If a voter detects a fault, it switches the role of couples: the master becomes slave and vice/versa. The system is thus operating normally as long as both processes in the master or in the slave work normally.

2. Software components allocation: it consists in mapping components of the software model onto components of the hardware model: each process must be assigned one processor to execute its threads, and each message between threads on different processors must be assigned one bus to be transmitted.

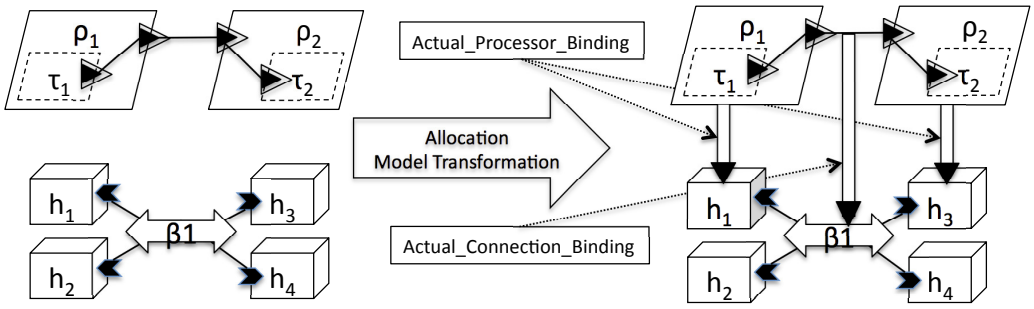

Figure 7.5: Example of allocations of software components

Figure 7.5 illustrates a simple transformation implementing the allocation of software components on hardware components. On the left part of this figure, the source AADL model is made up of two process components (ρ_1, ρ_2) and two processor components (h_1, h_2). In this model, software components are not yet allocated to hardware components. On the right part of the figure, target AADL model of the transformation shows the allocation of processes on processors (using the `Actual_Processor_Binding` property of AADL), and the allocation of messages on buses (using the `Actual_Connection_Binding` property of AADL).

7.3.2.3 Experimental results

Before applying our approach, we first describe the analysis methods we use to evaluate the NFPs of this case study: end-to-end response time and reliability.

1. End-to-end response time analysis: for a given path (see definition 9), end-to-end response time represents the highest amount of time required for an information to complete the path from the source to the destination. End-to-end response time takes into account both the worst-case response-time (WCRT) of threads and messages on the path. To compute these WCRTs, we use the response time analysis presented in [68].

$$R_{\tau_i,h_j} = \omega_{\tau_i,h_j} + \sum_{\tau_k \ s.t. \ \pi_{\tau_k} > \pi_{\tau_i})} \left\lceil \frac{R_{\tau_i}}{\varphi_{\tau_k}} \right\rceil * \omega_{\tau_k,h_j} + B_{\tau_i,h_j} \tag{7.1}$$

Response time of threads (Processor scheduling). Worst case response time R_{τ_i,h_j} of a periodic thread τ_i, executed on processor h_j, is represented with R_{τ_i,h_j} and computed according to (7.1). B_{τ_i} is the blocking time of the thread τ_i. It is computed using the access to the same shared resources for threads with lower priority (π_{τ_i} represents the priority of a thread τ_i). In our study, messages transmitted among tasks on the same processor represent shared resources and are protected with the Priority Ceiling Protocol (PCP) [69]. To compute the blocking time with the PCP we refer to equation (7.2), where $\omega'_{\tau_i,\mu_k,h_j}$ is the WCET of a thread τ_i for accessing (reading/writing) a shared message μ_k, executing critical section on h_j. The priority ceiling of this message is $PC(\mu_k)$.

$$B_{\tau_i,h_j} = \max_{\mu_k \in M, \tau_j \in T} \omega'_{\tau_i,\mu_k,h_j} \; s.t. \; \pi_{\tau_i} \leq PC(\mu_k) \; and \; \pi_{\tau_i} > \pi_{\tau_j} \tag{7.2}$$

Response time of messages (Bus scheduling). Worst case response time R_{μ_i,β_j} of message μ_i transmitted on bus β_j is computed according to equation (7.3). This equation is similar to equation (7.1) except that scheduling of messages is non-preemptive: when μ_i starts being transmitted, it cannot be interrupted. In the case of messages, blocking time B_{μ_i,β_j} accounts for the transmission, on μ_i, of at most one message (the one with the highest transmission time in the worst case) with a lower priority than μ_i. As a consequance, and according to [70], the blocking time of a message μ_i is computed by equation (7.4).

$$R_{\mu_i,\beta_j} = \omega_{\mu_i,\beta_j} + \sum_{\mu_k \; s.t. \; \pi_{\mu_k} > \pi_{\mu_i})} \left\lceil \frac{R_{\mu_i,\beta_j} - \omega_{\mu_i,\beta_j}}{\varphi_{\mu_k}} \right\rceil * \omega_{\mu_k,\beta_j} + B_{\mu_i,\beta_j} \tag{7.3}$$

$$B_{\mu_i,\beta_j} = \max_{\mu_j \in M} \omega_{\mu_j,\beta_j} \; s.t. \; \pi_{\mu_i} > \pi_{\mu_j} \tag{7.4}$$

End-to-end response time computation. The worst case end-to-end response time L_{Γ_i} is computed for each system path Γ_i. It consists in adding the WCRTs (see equation 7.1) of all threads and messages, as well as the periods of all the messages and their receiver thread on the path (see equation (7.5))[71].

$$L_{\Gamma_i} = \sum_{\tau_j \in \Gamma_i} R_{\tau_j} + \sum_{\mu_j \in \Gamma_i} R_{\mu_j} + \varphi_{\mu_j} + \varphi_{Dest(\mu_j)} \tag{7.5}$$

2. End-to-end reliability analysis: in order to compute reliability, we consider a simple fault model: only processor components may introduce fail-silent faults (*e.g.,* processor crashes). In our hardware models, each processor is characterized by its reliability: a constant value representing its probability to perform its operations as intended. $R(h_i)$ is the reliability of a processor h_i. End-to-end reliability is the probability, for a given

source of messages τ_s, to have its messages reach a destination τ_d. To compute end-to-end reliability, noted $R(\tau_s,\tau_d)$, we need to consider the reliability of paths (see definition 9) in the Γ. In order to compute the reliability of a path Γ_j, we have to consider the set of processors Γ_j traverses. A path Γ_j traverses a processor h_i if there exist a task τ_k in Γ_j such that $Proc(\tau_k)$ is allocated to h_i. We note $PS(\Gamma_j)$ the set of processors traversed by a path Γ_j. The reliability of $PS(\Gamma_j)$, noted $R(\Gamma_j)$, represents the probability that all the processors in $PS(\Gamma_j)$ perform their functions as intended. It is thus computed as follows:

$$R(\Gamma_i) = \prod_{h_i \in PS(\Gamma_j)} R(h_i) \tag{7.6}$$

Given the replication transformations introduced in previous subsection, the target models of these transformations have several paths with the same source and destination. In these models, we note $SE(\Gamma_{s,d})$ the set of paths having τ_s as source and τ_d as destination.

If paths of $SE(\Gamma_{s,d})$ result from the application of the 2oo3 transformation, $SE^{2oo3}(\Gamma_{s,d})$ is made up of three paths: $SE^{2oo3}(\Gamma_{s,d}) = \{\Gamma'_{s,d}, \Gamma''_{s,d}, \Gamma'''_{s,d}\}$. Besides, messages from τ_s reach τ_d if at least two out of these paths perform as intended. We thus obtain the following formula:

$$\begin{aligned} R^{2oo3}(\tau_s,\tau_d) = & \prod_{\Gamma_i \in SE^{2oo3}} R(\Gamma_i) \\ & + R(\Gamma'_{s,d}) * R(\Gamma''_{s,d}) * (1 - R(\Gamma'''_{s,d})) \\ & + R(\Gamma''_{s,d} * R(\Gamma'''_{s,d}) * (1 - R(\Gamma'_{s,d})) \\ & + R(\Gamma'_{s,d}) * R(\Gamma'''_{s,d}) * (1 - R(\Gamma''_{s,d})) \end{aligned} \tag{7.7}$$

To compute the reliability $R^{2*2oo2}(\tau_s,\tau_d)$, we proceed in a similar way by evaluating the reliabilities of two pairs of paths (master and slave).

Once all the input artefacts (the architecture model, the model transformation chain, NFPs analysis methods) are defined, we configure our approach to (i) identify for each link of the model transformation chain confluent *TRIs* and atomic transformation alternatives, and (ii) optimize (using EAs) the model transformation chain to produce optimal (or near optimal) architecture alternatives (as presented in section 6.3).

In subsection 7.3.2.1, we gave a general definition of the used models without specifying the number of process and processor components. For this experiment we use a source model composed of 3 processes and 5 multi-core processors. So that applying a safety design pattern (i.e., 2*oo*3 or $2*2oo2$) the software architecture is made up of 9 to 12 pro-

cesses. In addition, the hardware platform is sophisticated enough to provide an important set of possible allocations: 20 for each replicated process. Thus, this experiment provides a very important design space.

In addition, we configure the NSGA-II algorithm as follows: the size of the population is set to 100 composite transformations CTs, the crossover probability is set to 0.9, the mutation probability is set to $1/nATA$ (nATA being the number of atomic transformation alternatives $ATAs$ in a CT), the maximum number of iterations is set to 10000 (10 for the replication transformation, 1000 for the allocation transformation), and the maximum number of iterations without finding new non-dominated solutions is set to 5.

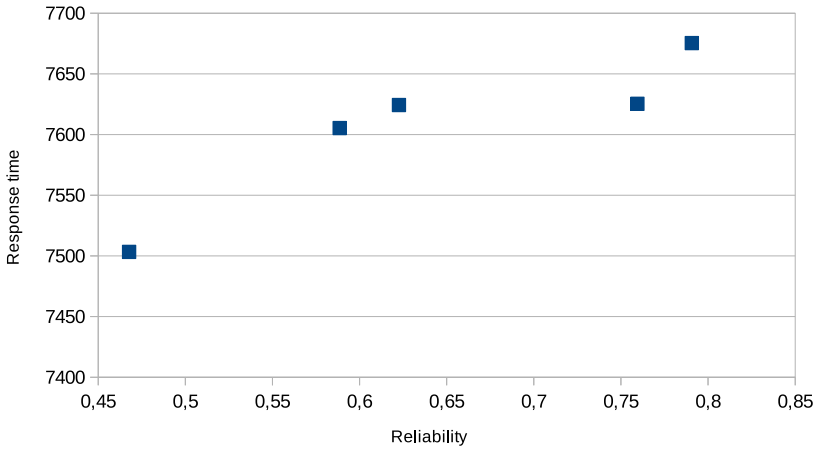

Figure 7.6: Experimental results for the second case study

Figure 9.7 shows that our approach could be used to exhibit a set of 5 solutions, each of them presenting non-dominated solutions in terms of reliability and response time. Thus, the designer can select the solution which suites him best.

To summarize, we applied our framework on two different optimization problems (model refinement and replication/allocation of software components), and from the experimental results we conclude that our approach can be applied to resolve different optimization problems. These results demonstrate the genericity of our framework.

7.4 Evaluation of the quality of the framework

In order to evaluate the quality of the proposed approach, we need to answer to the remaining research questions:

- How efficient is it to proceed to structural constraints validation a priori instead of a posteriori?
- What is the distance between results obtained with our approach and theoretical local optimum?
- How much resource (memory and computation time) requires our approach to explore numerous composite transformations?

To answer these questions, we perform some experiments on the case study defined in subsection 7.3.2. More precisely, we define 4 AADL models with different levels of complexity (e.g., different number of processes, connections among processes). Then, we apply our framework on these models, and make our observations. The characteristics of the AADL models are listed in table 7.2, allowing to estimate the complexity of the design space. For instance, *model 1* is made up of 3 paths (see definition 9), which leads to $2^3 = 8$ composite transformations for the replication. For the allocation, the worst situation occurs when the 2*2oo2 replication is applied to each process in the input model: with *model 1* this leads to an allocation problem with $9*4 = 36$ processes on 5 processors. Considering allocation constraints, a rapid estimation of the complexity leads to 12960 (creating systematically 3 replicas in the first transformation link) to 34520 possibilities (creating systematically 4 replicas in the first transformation link). With *model 4*, the number of potential architectures is between 2 to 20 millions. Note that *model 3* is a significant subset of an industrial use-case from the railway domain, dealing with doors control in an automatic train.

Table 7.2: Input models for experimentations

model identifier	Number of processes	Number of paths	Number of processors
model 1	9	3	5
model 2	8	4	8
model 3	9	4	10
model 4	15	6	10

In addition, we configure the NSGA-II algorithm with the same properties defined in subsection 7.3.2.3: the size of the population is set to 100, the maximum number of iteration is set to 10000, and the maximum number of iterations without finding new non-dominated solutions is set to 5.

Applying our approach using the defined input artefacts (4 AADL models, safety/allocation model transformations, and the configured NSGA-II), we obtain some experimental results. Using these results, we answer to the research questions.

7.4.1 A priori vs. a posteriori validation

Evaluation method. We compare the convergence time required by our approach with the convergence time required by an approach based on EAs but relying on a posteriori validation of structural constraints.

Table 7.3: Comparison of validation techniques

Validation technique	Number of iterations	Number of optimal solutions	Convergence time
A posteriori	2000	0	Not converged
A priori	2000	3	Converged after 11 hours

Results. Table 9.1 presents the results of the execution of our approach on the source model 1 (see table 7.2) using both validation techniques. Out of two thousand candidate architecture generated by the approach based on **a posteriori** validation, none of them is correct with respect to structural constraints. Thus, this approach do not converge and no optimal (or near optimal) solutions are found.

On the other hand, the approach based on a priori validation is able to converge and identify 3 optimal architectures by evaluating more than two thousand correct architectures in 11 hours of computation. This shows the importance of validating composite transformations **a priori** rather than a posteriori.

7.4.2 Distance to local optimum

Evaluation method. We compare the distance between solutions obtained with our approach and a theoretical local optimum. More precisely, we formalise this distance in terms of percentage of the maximum distance between a best theoretical local optimum, and a worst theoretical local optimum.

To quantify the distance between results obtained with our approach and theoretical local optimum, we first need to develop an approach capable to find local optimum. Considering a subset of the design space to explore, we implement a Mixed Integer Linear Program (MILP) formulation. Linear programming techniques such as MILP are known to be very efficient. It is thus a very good reference method. Thus, we use the MILP formulation to compute local optimum corresponding to the best local optimum, and the worst local optimum: the best local optimum is found by maximizing an objective function, while the worst is found by minimizing the objective function.

However, note that the solution based on a MILP required to linearize the multi-objective problem defined in section 7.3.2 into a single objective function. We define this objective function as a weighted sum of the following objectives: reduce response-time

and increase reliability. Note that weights must be set manually, and are quite difficult to set in order to produce different solutions. We first set the weight with extreme values (0.00001 for reliability and 0.99999 for response time, and vice-versa) to find at least two possible solutions. Then, we try to find new solutions by dichotomy. This process is long, and result in very partial results for this method.

Additionally, the MILP-based solution requires additional adjustments that are made manually: we have to linearise the function that computes reliability. To do so, we have to consider restrictive hypothesis: (i) all the processors in the model of the hardware platform have the same reliability *Rel*, and (ii) if a set of inter-connected processes $\rho = \rho_i$ is replicated into $\rho' = \rho'_i$, $\rho'' = \rho''_i$, etc. then all the processes of ρ' must be allocated on the same processor, all the processes of ρ'' must be allocated on the same processor, etc. Under these restrictive hypothesis, we evaluate the choice of a safety design pattern in terms of reliability: the choice of $2oo3$ for a path leads to a reliability of $Rel^3 + 3 * Rel^2 * (1 - Rel)$ for this path, while the choice of $2 * 2oo2$ leads to a reliability of $1 - (1 - Rel^2)^2$. These values are stored as constant values in the MILP. As a consequence, the MILP considers only a subset of the possible architectures.

In addition, MILP is not applicable if processors of different reliability are used: we avoid this situation in our experiments in order to be able to obtain comparative results. Thus, the implemented MILP represents a simplified version of the used evolutionary algorithm.

Results. Comparing results obtained with MILP and our approach, we come to the following conclusions: imposing the allocation constraints of the MILP formulation in our approach, solutions found with the MILP formulation dominate those found by our approach. Measuring the relative distance between solutions we found and solutions found by MILP, we observe for the two first models that the distance is very low (less than 3%) while it increased for models 3 (24,5%) and model 4 (34%). This is typically due to the fact that the percentage of the explored design space becomes smaller and smaller as the complexity increases.

Without imposing the allocation constraints of the MILP formulation in our approach, we observe that our method find solutions that are not dominated by solutions find in MILP. These solutions are typically better in terms of reliability than solutions found by the MILP. Compared to a random exploration, our approach converged after evaluating 1920 (considering *model 1*) to 2121 (considering *model 4*) output models (see table 9.2. Random exploration do not converge and stop after evaluating 5000 to 10000 output models. No solution found with the random method dominate solutions found in our approach. On

the other hand, some solutions found with our method dominated solutions found with the random method.

As a conclusion, our experimentations show that our method find solutions (i) relatively close to local optima, and (ii) not dominated by local optima. In addition, it finds better solutions much faster than a random exploration.

7.4.3 Resources consumption

Evaluation method. To evaluate the quantity of resources required by the execution of our method, we measure the peaks of memory space and computation power it was using at runtime, as well as the time it need to converge.

Table 7.4: Measured resource consumption

model identifier	Convergence time	Evaluated architectures	memory usage
model 1	11h09m	1920	13 GB
model 2	9h39m	2121	12.4 GB
model 3	13h24m	1710	14 GB
model 4	14h23m	1415	13 GB

Results. Table 9.2 presents the measured resource consumption, at runtime, for each of the input models we considered. Experiments were done using a parallel execution of EAs iterations on 22 cores computer. These results show high resource consumption in the different models. Measuring the time spent in different parts of our implementation, we could identify that an important part of the computation time (more than 50%) is spent executing internal transformations produced by higher order transformations (HOTs): a HOT is a model transformation that takes as source and/or produce as target model transformations [56]. Our framework indeed executes HOTs to merge transformation alternatives into a single transformation which implements the selection of transformation rule instantiations *TRIs*. This solution was initially made not to modify the ATL runtime, but it appears to be very time consuming in practice. Consequently, we believe the execution time we measure would be significantly improved by re-implementing this part of our framework.

7.4.4 Threats to validity

We discuss several threats to validity of the results presented in the previous subsection.

As far as the usability of our approach is concerned, we acknowledge that existing model transformation repositories do not provide model transformation alternatives. As such, these transformations are not intended to be used in a design space exploration process. To use model transformations in a design space exploration process, developers

must identify valuable transformation alternatives. Our approach helps to formalize such alternatives and provides a composition method to proceed to design space exploration.

As far as experimental results are concerned, we acknowledge that the necessity to use **a priori** validation of composite transformations depends on the ratio of correct vs. incorrect architectures in the design space produced when ignoring validity constraints. In our experience, when architecture alternatives are precisely specified with model transformations, such constraints rapidly make the ratio of correct vs. incorrect solution rather small. This was already the case in the experiment presented in section 7.3.2.

We also acknowledge that these experimental results rely on a small set of input models. However, one of them is extracted from an industrial case-study. The most complex models we considered represent software architectures of a reasonable complexity in the domain of real-time embedded systems. In addition, the replication patterns we use are replication patterns used in industry, and their specification using ATL model transformations is quite complete. It is thus difficult to generalize the results obtained in subsection 7.4 to other models and design alternatives, but the experimental results we provide already demonstrate the feasibility of our approach on significant design problems.

7.5 Conclusion

In this chapter we have detailed the experimental validation and evaluation of the main contributions of this thesis. First we formulated a set of research questions in order to evaluate the genericity and quality of our approach. More precisely, these research questions were used to validate a set of assumptions made during this thesis: (i) applying our approach in different optimization problems, (ii) validating a priori structural constraints of composite transformations is better than an a posteriori validation, and (iii) applying evolutionary algorithms on a design space of composite transformations is more efficient than other design space exploration algorithms.

To check the genericity of our approach, we first described different railway case studies. Then, we applied different design decisions (model refinement, safety patterns and software component allocation) on these case studies. Afterwards, for each case study we applied our framework in order to find a set of optimal (or near optimal) architecture alternatives. Our conclusion from the obtained results is that our approach can be reused on different optimization problems.

To check the quality of our approach, we performed a comparison between (i) the application of our framework using an a priori validation of composite transformations, and (ii) the application of our framework using an a posteriori validation of composite

transformations. We stated that the framework based on an a posteriori validation (i) is time consuming, and (ii) does not converge to optimal (or near optimal) architecture alternatives. In addition to that, we proposed to (i) linearize a multi-objectives problem into a single objective function defined as the weighted sum of existing objectives, (ii) compute an optimal solution using linear programming techniques, (iii) evaluate the results obtained with our approach; and (iv) compare the results of our approach with the results of the application of the linear programming version. To perform this comparison we applied both our approach and the linear programming version on several system models.

All the experimental results validate the capability of our method to explore a design space of model transformation chains and identify a set of optimal (or near optimal) architectural alternatives with conflicting NFPs.

8 Conclusion and Future work

CONTENTS

8.1 Conclusion

In this thesis we described an approach to explore a design space of architectural alternatives, and select the alternatives answering at best to a trade-off among conflicting non functional properties. Exploring such design spaces presents several problems. Large and complex systems may admit millions of components and interactions among these components. Enumerating and modelling all the possible architecture alternatives is error-prone and sometimes impossible to perform. Thus, an automated solution to identify and model the design space of architecture alternatives is necessary. Additionally, not all the produced architecture alternatives are correct regarding different structural constraints, and designers have to check the correctness of each alternative before exploring them. Performing this operation is costly and laborious, and in some cases the designers may not find any optimal (or near optimal) architecture alternative. Finally, to explore a design space of architecture alternatives involving multiple conflicting non functional properties, multiple-objectives optimization techniques (MOOTs) are of great interest. However, MOOTs are based on heuristics such as variation operators. These operators are usually applied on solutions formalized using a binary or real encoding. Thus to apply MOOTs on a design space of architecture alternatives a specific encoding of the alternatives is needed.

To solve these problems, we propose a generic framework that combines multi-objective optimization techniques and model transformation techniques. The main contributions of this thesis are summarized in the following paragraph.

The first contribution of this thesis consists in automating the formalization and analysis of architecture alternatives. In our approach, architecture alternatives are automatically generated using ATL, a rule-based model transformation language. We consider this type of model transformations for different reasons. Rule-based model transformations can be reused for different multiple-objective optimization problems without modifying the exploration engine. Thus, our framework is generic and can be applied to solve different optimization problems. The execution of a rule-based model transformation produces transformation rule instances that can be composed together to create new transformations (called composite transformations), and thus automate the generation of different architecture alternatives. Additionally, the correctness of architecture alternatives regarding the structural constraints can be checked a-priori (before the application of a composite transformation) using boolean formulas and SAT solving techniques. This allows to exclude composite transformations producing incorrect architectures, and thus reduce the design space to explore. Finally, complex optimization problems can be formulated, and explored by composing externally (i.e., chaining) two or more rule-based model transformations. However, the design space grows rapidly with the number of composite transformations, which leads to a combinatorial problem.

Facing the problems raised by architecture exploration, and in particular the combinatory explosion of composite transformations and the conflicting character of NFPs, we automated the production of composite transformations using a well-known multiple-objective optimization technique: Evolutionary Algorithms (EAs). In this thesis, EAs take as input an initial population of randomly created composite transformations, and apply different steps (evaluation, selection and genetic operators) to create new composite transformations. These steps are repeated a certain number of times until producing a set of composite transformations which generate optimal (or near optimal) alternatives. More formally, the application of EAs on a design space of composite model transformations requires to structure these transformations in order to guarantee that genetic operators (e.g., mutation, crossover) are correctly applied. In our approach, we propose to structure composite transformations as ordered sets of atomic transformation alternatives. Note that, these sets are produced by applying the a-priori validation: identification of confluent transformation rule instantiations and SAT solving. Thanks to this structure, we can alter (crossover, mutation) atomic transformation alternatives of composite transformations and always produce new valid composite transformations.

In order to consider more complex optimization problems, and also to facilitate the maintainability and reusability of model transformations we proposed to chain two or more rule-based model transformations (i.e., the output model of a model transformation

becomes the input model of a new model transformation). Thus, we generate architecture alternatives by executing model transformation chains. Therefore, to identify optimal (or near optimal) architecture alternatives, we need to explore design spaces composed of model transformation chains. To reach this objective, we applied EAs on each model transformation (also called link when applied in a chain) of a given chain. More precisely, we iterated over model transformations of a chain and produced the most suitable composite transformations for each link.

To validate the applicability and the efficiency of our approach, we executed our exploration engine on two different experimental case-studies, and checked the experimental results regarding different criteria. These criteria aim at evaluating (i) the genericity of our approach, (ii) the resource (memory and computation time) consumption of our approach, (iii) the efficiency of our exploration engine compared to other possible exploration solutions such as linear programming. All the obtained experimental results validate the capability of our framework to explore a design space of composite transformations (and model transformation chains), and identify a set of optimal (or near optimal) architectural alternatives with conflicting NFPs.

8.2 Future Work

The validation of each of our contributions uncovered remaining open points for which we have identified promising solutions to be investigated:

Determining the parameters to use for a given optimization problem is critical to the success of evolutionary algorithms. Using good parameters helps in identifying more quickly optimal solutions. Usually, these parameters (e.g., probability of crossover, probability of mutation, size of the population) are determined by trial-and-error. When performing our experiments, we used standard parameters (e.g., fixed mutation and crossover probabilities) for NSGA-II. These parameters were used in several experimental setups we found in research works evaluating optimization frameworks. These parameters are good enough to validate the applicability of our exploration framework. In some works [72; 73], the authors proposed to vary dynamically the parameters of evolutionary algorithms regarding the results of the candidate solutions in each generation. For instance, the probability of applying the crossover (and mutation) increases (or decreases) depending on the results obtained in the previous application of the crossover (and mutation). From their results, the authors stated that varying the parameters of an EA helps in identifying more quickly optimal solutions. We plan to incorporate these works in our exploration engine in order to evaluate the sensitivity of our exploration framework with respect to variations

in the parameter of the EA, and thus converge more quickly in optimal (or near optimal) architecture alternatives.

In our experiments, we considered model transformation chains composed of only two rule based model transformations: safety pattern and software components allocation. We plan to raise the scope of application of our exploration engine by considering more complex model transformation chains composed of more than two rule-based model transformations. For example, we consider a chain composed of four design patterns: software component replication, software component allocation, components selection, and change of hardware and software parameters. We also plan to consider other model transformation chains related to other domain fields, and not only railway.

When applying our exploration engine on different case studies, we observed that our framework spent a lot of time (more than 50 % of the computation time) in composing internally (using HOTs) transformation rule instantiations, and thus produce composite transformations. In our experiments, we considered case studies composed of 8 to 15 interconnected software components, and 5 to 10 hardware components. For large and more complex case studies, the convergence of our exploration framework will be very impacted by the creation of composite transformations. Consequently, we plan to improve the performances of our architecture exploration framework in order to scale to even more complex case-studies. In particular, we plan to implement a dedicated ATL runtime in order to reduce the computation time overhead due to the usage of internal composition (HOTs).

8.3 Acknowledgements

The work described in this thesis has been carried out in the framework of the Technological Research Institute SystemX, and therefore granted with public funds within the scope of the French Program “Investissements d’Avenir”. It was also partially funded by the academic and research chair Engineering of Complex Systems.

9 Résumé de la thèse en Français

CONTENTS

Ce document résume les travaux effectués dans le cadre de la thèse de Smail RAHMOUN sous la supervision de Laurent PAUTET et Etienne BORDE de Télécom ParisTech. Cette thèse a été effectuée au sein de l'Institut de Recherche Technologique SystemX (IRT SystemX) dans le cadre du projet FSF (Fiabilité et Sûreté de Fonctionnement). Ce projet a regroupé de nombreux acteurs du secteur industriel, comme ALSTOM Transport, APSYS, Esterel Technologies, Krono-Safe, Scaleo Chip. Ce projet a aussi regroupé de nombreux acteurs du secteur académique tels que CEA, Inria, Institut Mines-Télécom, Université Paris-Sud.

9.1 Contexte

Prédire les propriétés non fonctionnelles d'un système logiciel avant sa mise en œuvre est crucial pour le succès du processus de développement. Cela permet aux concepteurs logiciels d'anticiper les défaillances potentielles du système, d'éviter des maintenances coûteuses, et ainsi créer des systèmes plus performants.

La prédiction des propriétés non fonctionnelles (PNFs) des systèmes logiciels a été un domaine de recherche majeur ces dernières années. Plusieurs travaux ont proposé différentes méthodologies et outils pour estimer différentes PNFs telles que la performance, la fiabilité, le coût, la disponibilité, la fiabilité, etc. Ces travaux s'appuient sur des techniques de modélisation. En effet, les modèles présentent plusieurs avantages. Ils offrent une vue plus compréhensible d'un système en cours de développement. Ils mettent en évidence les artefacts les plus importants (les composants matériels, la mémoire, les tâches, etc.) d'un système donné ainsi que leurs interactions. En plus de tout cela, les modèles offrent la possibilité d'analyser des systèmes par rapport à différentes propriétés fonctionnelles et non fonctionnelles avant qu'une implémentation concrète soit produite. Ainsi, à l'aide de modèles, les concepteurs systèmes peuvent vérifier si un système donné est censé répondre aux PNFs requises.

En compléments, des patrons de conception ont été proposés pour améliorer les PNFs d'un système. Les patrons de conception sont des solutions réutilisables permettant de résoudre des problèmes de conception récurrents. Il est donc utile d'avoir connaissance de ces patrons de conception afin d'envisager différentes options de conception, et ainsi analyser et améliorer différentes PNFs.

Le projet FSF (Fiabilité et Sûreté de Fonctionnement) est un bon exemple de projet où les concepteurs ferroviaires veulent prédire les PNFs d'un système embarqué avant sa mise en œuvre. Ce projet est basé sur un cas d'utilisation ferroviaire où les concep-

teurs d'Alstom Transport souhaitent organiser les ressources logicielles et les ressources matérielles afin que le système résultant améliore l'ensemble de PNFs suivantes: sûreté, fiabilité, disponibilité et temps de réponse.

Plus précisément, dans ce projet, les concepteurs ferroviaires sont intéressés par l'application de patrons de conception (réplication logicielle, allocation de composants logiciels, raffinement de modèles) dans le développement de systèmes embarqués critiques. L'application de ces patrons sera évaluée en fonction de leur impact en termes de sûreté, de fiabilité, de temps de réponse, et de maintenabilité, etc. Par exemple, nous considérons le patron de conception permettant de tripliquer des composants matériels. Ce patron consiste à reproduire un composant matériel trois fois. Il utilise également un voteur pour (i) collecter les sorties des trois répliques, et (ii) déterminer la sortie finale réelle du système. Ainsi, le voteur représente un artefact de prise de décision. Cependant, l'utilisation d'un voteur augmente le temps de réponse du système, car le voteur communique avec chaque réplique matérielle et utilise des fonctions pour déterminer la sortie finale du système. D'autre part, en raison de l'utilisation de trois répliques matérielles le système est plus sûr. A noter aussi qu'on peut utiliser différents patrons de conception permettant de répliquer un certain nombre de fois des composants matériels. Ainsi, différentes alternatives architecturales peuvent être produites à partir de l'application de ces patrons. Ces architectures sont ensuite évaluées en termes de sûreté et de temps de réponse afin de trouver l'architecture qui maximise l'ensemble des PNFs.

Dans le projet FSF, nous proposons de (i) créer l'espace de conception des alternatives architecturales en utilisant des patrons de conception, (ii) évaluer les alternatives architecturales par rapport à différentes PNFs, et (iii) comparer les alternatives afin de trouver l'architecture optimale.

9.2 Défis

Le but de cette thèse est de développer une approche basée sur les modèles qui contribue à la prédiction des propriétés non fonctionnelles (PNFs) d'un système logiciel. Pour développer une telle approche, nous devons surmonter les défis suivants.

Dans de nombreux cas, les PNFs sont conflictuelles: l'amélioration d'une PNF peut avoir un impact négatif sur d'autres PNFs. Par exemple, l'amélioration de la disponibilité d'un composant matériel se fait souvent par la réplication de ce composant, ce qui augmente le poids du système ainsi que le temps de réponse du composant répliqué. De même, l'allocation de plusieurs composants logiciels inter-connectés sur le même composant matériel améliore le temps de réponse du système. Pourtant, centraliser tous les

composants logiciels dans un seul composant matériel diminue la fiabilité du système. Dans ces exemples, l'amélioration d'une PNF diminue les autres PNFs. Ainsi, la modélisation d'un système qui améliore tous ses PNFs simultanément n'est pas possible dans le cas où les PNFs sont conflictuelles. Par conséquent, les concepteurs doivent envisager plusieurs alternatives architecturales, et doivent les explorer pour trouver les alternatives qui optimisent les PNFs. En d'autres termes, les concepteurs sont confrontés à un problème d'optimisation multiple-objectifs (POMO).

Un POMO est caractérisé par (i) un espace de conception d'alternatives architecturales, et (ii) des fonctions objectifs qui évaluent la qualité de ces alternatives. Les fonctions objectifs de chaque alternative dans l'espace de conception sont calculées pour évaluer la qualité de l'alternative. Dans cette thèse, les objectifs représentent des PNFs. Étant donné que les PNFs sont conflictuelles, il n'existe pas une unique alternative architecturale qui est optimale mais une variété de choix qui représentent différents compromis entre les PNFs. L'optimalité dans ce cas est définie en utilisant le concept de Pareto-dominance: une alternative architecturale domine une autre si elle est égale ou meilleure dans tous les objectifs et strictement meilleure dans au moins un objectif. Dans un espace de conception d'alternatives architecturales, celles-ci sont appelées solutions Pareto-optimales, elles représentent les alternatives architecturales qui dominent toutes les autres alternatives. Ainsi, la résolution d'un POMO consiste à trouver les alternatives architecturales Pareto-optimales.

Évaluer chaque alternative architecturale par rapport à des PNFs est une opération impossible en pratique, car l'espace de conception peut rapidement devenir très important. Par exemple, considérons un système embarqué contenant 30 composants logiciels, et 10 composants matériels. Chaque composant logiciel peut être alloué sur n'importe quel composant matériel, conduisant à un espace de conception composé de 1030 alternatives architecturales. La complexité augmente encore plus quand il y a plusieurs PNFs en conflit. Dans ce cas la, chaque alternative architecturale est évaluée par rapport aux PNFs et les résultats de l'évaluation doivent être comparés avec les résultats d'évaluation de chacune des autres alternatives. Ce processus est appelé exploration de l'espace de conception.

L'exploration de l'espace de conception est un processus permettant d'explorer un espace de recherche composé d'alternatives architecturales, et d'identifier les solutions Pareto-optimal. Au cours des années précédentes, plusieurs travaux ont proposé des solutions à ce problème. Ces solutions sont basées sur des stratégies de recherche qui (i) vérifient l'exactitude des alternative architecturale et rejettent les solutions incorrects tout en explorant l'espace de conception, (ii) évaluent les alternatives correctes concernant dif-

férentes PNFs, et (iii) comparent les alternatives architecturales afin d'identifier les alternatives Pareto-optimal. Ces solutions d'exploration sont simples à mettre en œuvre, mais elles sont très coûteuses: plus la taille de l'espace de conception augmente, plus il devient difficile d'explorer l'espace de conception tout en rejetant les solutions incorrectes. De plus, dans ces travaux, pour chaque nouveau problème d'optimisation multi-objectifs, la stratégie d'exploration utilisée doit être adaptée au nouveau problème afin de produire les alternatives architecturales et les explorer. En d'autres termes, ces solutions ne sont pas génériques en ce qui concerne les différents problèmes d'optimisation multiple-objectifs.

Dans cette thèse, nous voulons fournir un framework d'exploration automatisé et générique qui nous permettra de résoudre différents problèmes d'exploration d'espaces de conception impliquant de multiples PNFs conflictuelles.

9.3 Problématique

Créer un framework générique et automatisé permettant d'explorer un espace de conception composé d'alternatives architecturales présente les problèmes suivants:

9.3.1 Création de l'espace de conception

Les concepteurs logiciels réutilisent des patrons de conception existants pour produire des alternatives architecturales. Souvent, ces patrons sont appliqués manuellement pour créer les alternatives architecturales. Afin d'automatiser l'application des patrons de conception, l'Ingénierie Dirigée par les Modèles (IDM) a introduit le concept des transformations de modèle. Les transformations de modèles formalisent dans des artefacts réutilisables l'implémentation de différents patrons de conception tels que des décisions de conception, des raffinements de modèles.

Une transformation de modèle spécifie un ensemble d'actions pour générer un modèle cible à partir d'un modèle source. Le but sous-jacent à l'utilisation des transformations du modèle est de gagner du temps et des efforts et de réduire les erreurs en automatisant la production et les modifications des modèles. De plus, les transformations de modèle peuvent être réutilisées dans différents projets ou produits sans modifications majeures. Dans ce contexte, les alternatives de transformations de modèle sont des transformations qui produisent différents modèles cibles lorsqu'elles sont appliquées sur le même modèle source. Ces modèles cible représentent des alternatives architecturales qui peuvent différer dans les PNFs (par exemple fiabilité, temps de réponse, etc.) qu'ils présentent. Les alternatives de transformation peuvent être composées (enchaînées ou fusionnées) ensem-

ble pour créer de nouvelles transformations de modèle, et ainsi produire automatiquement de nouvelles alternatives architecturales. Une transformation de modèles est une solution prometteuse afin de modéliser les alternatives architecturales pour les différents problèmes de conception. Cela représente une solution générique qui peut être incorporée dans un framework d'exploration. Cependant, l'espace de conception des alternatives architecturales croît rapidement avec le nombre de transformations de modèle créées et d'éléments dans le modèle, ce qui conduit à un problème combinatoire. Ainsi, trouver les alternatives architecturales optimales à partir de toutes les alternatives réalisables devient difficile à réaliser en utilisant seulement la composition des transformations de modèle.

9.3.2 Vérifier l'exactitude des alternatives architecturales

Lors de l'exploration d'un espace de conception d'alternatives architecturales un autre problème est de vérifier l'exactitude des alternatives architecturales en fonction de différentes contraintes structurelles (les contraintes de mémoire, la co-localisation des composants logiciels, etc). Les contraintes structurelles sont définies par les concepteurs logiciels et sont nécessaires pour guider l'exploration de l'espace de conception vers des alternatives architecturales correctes.

Dans le passé, plusieurs travaux ont proposé des solutions pour vérifier l'exactitude des alternatives architecturales par rapport à un ensemble de contraintes structurelles. Dans ces travaux, les contraintes structurelles sont vérifiées tout en explorant les alternatives architecturales: évaluation et comparaison des alternatives architecturales. Plus précisément, les contraintes structurelles sont exprimées sur chaque alternative architecturale en utilisant un langage de spécification de contraintes. Ce langage est alors utilisé pour vérifier l'exactitude de l'alternative architecturale. Si l'alternative respecte les contraintes structurelles, cette alternative est considérée dans l'espace de conception à explorer et est évaluée et comparée à d'autres alternatives correctes. Dans le cas contraire l'alternative architecturale est rejetée. Cette opération peut prendre très longtemps pour trouver des alternatives correctes. Dans certains cas, cette opération peut ne pas converger du tout (pas d'alternative architecturale correcte). Ainsi, nous devons fournir un framework d'exploration qui permet d'identifier les alternatives architecturales correctes tout en réduisant le temps de recherche, et en convergeant vers des alternatives architecturales optimales (ou proches d'architecture optimale).

Dans la section précédente, nous avons proposé de créer l'espace de conception des alternatives architecturales en utilisant des compositions de transformation de modèle. Dans ce cas, il serait plus intéressant de vérifier les contraintes structurelles sur les transforma-

tions de modèle avant de les exécuter, et de produire ainsi les alternatives architecturales correspondantes. Cette solution permet de créer un espace de conception composé uniquement d'alternatives architecturales correctes. Cependant, un autre problème se pose lors de la vérification des contraintes structurelles sur les transformations de modèle. Les contraintes structurelles sont généralement exprimées sur des modèles en utilisant des langages de contraintes de haut niveau tels que Object Constraint Language (OCL). D'autre part, les transformations de modèles sont représentées par un code écrit dans un langage de programmation. Ainsi, nous devons trouver une solution pour exprimer des contraintes structurelles sur des compositions de transformation de modèle afin de produire des alternatives architecturales correctes.

9.4 Identification et composition de transformations de modèles

Le premier problème abordé dans cette thèse est apparu dans le contexte de l'analyse d'alternatives architecturales par rapport à un ensemble de propriétés non fonctionnelles (PNFs). En effet, développer un système logiciel qui améliore simultanément un ensemble de PNFs est complexe à réaliser. Cette complexité est en partie due au fait que les PNFs sont souvent en conflit: l'amélioration d'une PNF en particulier peut dégrader une ou plusieurs autres PNFs. Par conséquent, nous devons (i) identifier et formaliser plusieurs alternatives architecturales, (ii) analyser chaque alternative par rapport aux PNFs, et (iii) sélectionner les alternatives architecturales qui répondent au mieux au compromis entre les différentes PNFs.

Dans la section suivante, nous décrirons comment formaliser les alternatives architecturales afin de (i) faciliter leur analyse par rapport aux PNFs et (ii) faciliter leur exploration et leur sélection.

9.4.1 Modélisation des alternatives architecturales

Une solution pour formaliser les alternatives architecturales serait d'utiliser des modèles abstraits. Ces modèles décrivent un aspect du système qui n'est pas facilement ou suffisamment capturé par l'implémentation du système (comme les PNFs). Ainsi, en utilisant ces modèles, nous pouvons analyser les PNFs de chaque alternative architecturale. Afin d'automatiser la modélisation des alternatives architecturales, nous utilisons une technique d'ingénierie bien connue, appelée transformation de modèles.

Une transformation de modèle est un artefact logiciel qui spécifie un ensemble d'actions pour générer un modèle cible à partir d'un modèle source. Dans ce contexte, nous considérons que les alternatives de transformation de modèle sont des transformations de modèles qui génèrent des alternatives architecturales lorsqu'elles sont appliquées au même modèle source. Dans cette thèse, nous nous concentrons sur des alternatives de transformation de modèle produisant des architectures présentant différentes PNFs (par exemple fiabilité, temps de réponse, etc.).

Généralement, les transformations de modèle sont écrites dans un langage de transformation de modèles. Au cours des années précédentes, plusieurs langages de transformation de modèles ont été proposés pour écrire des transformations de modèles telles que des langages de transformation de modèles à base de règles, des langages déclaratifs. Dans cette thèse, nous proposons l'utilisation de langages de transformation de modèles à base de règles pour écrire nos transformations de modèle. Nous proposons d'utiliser ce type de transformations car elles utilisent une syntaxe claire basée sur des langages déclaratifs et impératifs. Elles fournissent également une sémantique d'exécution simple, et permettent de structurer les transformations et ainsi faciliter la génération d'alternatives architecturales.

Une transformation de modèle à base de règles est une transformation de modèle composée d'un ensemble de règles de transformations. Une règle de transformation spécifie la transformation des éléments du modèle source (par exemple méta-classes du méta-modèle source de la transformation), vers les éléments du modèle cible (par exemple, les méta-classes du méta-modèle cible de la transformation).

Plus formellement, l'application de transformations de modèle à base de règles instancie pour chaque règle de transformation, un ensemble d'instanciations de règles de transformation (IRTs). Une instanciation de règle de transformation IRT_i représente l'application d'une règle de transformation sur un ensemble ordonné d'éléments d'un modèle source. Elle peut être représenté comme un tuple $< R, E_i, A_i >$, où:

- R représente la règle de transformation appliquée;
- E_i représente le tuple d'éléments du modèle source sur lequel on applique une action A_i;
- A_i est l'ensemble des actions qui sont exécutées lorsqu'elles sont appliquées à E_i.

Pour illustrer cette définition d'IRT, nous présentons l'exemple suivant: une transformation de modèle à base de règles (appelée AllocationRule) composée d'une règle

de transformation qui permet d'allouer deux composants logiciels (Sw_1 et Sw_2) sur deux composants matériels Hw_1 et Hw_2.

En Appliquant la règle AllocationRule sur les éléments du modèle source (composants logiciels et matériels), nous avons crée quatre IRTs:

$$IRT_{11} =< AllocationRule; \{Sw_1; Hw_1\}; allouer(Sw_1, Hw_1) >;$$
$$IRT_{12} =< AllocationRule; \{Sw_1; Hw_2\}; allouer(Sw_1, Hw_2) >;$$
$$IRT_{21} =< AllocationRule; \{Sw_2; Hw_1\}; allouer(Sw_2, Hw_1) >;$$
$$IRT_{22} =< AllocationRule; \{Sw_2; Hw_2\}; allouer(Sw_2, Hw_2) >.$$

Dans ces IRTs, les actions de la règle sont notées **allouer(**Sw_i**,** Hw_j**)**. Cette action spécifie que Sw_i est alloué à Hw_j. Par exemple, IRT_{11} représente l'allocation de Sw_1 sur Hw_1, et IRT_{22} représente l'allocation de Sw_2 sur Hw_2.

L'exécution de chaque IRT transforme un tuple d'éléments du modèle source en leur image dans le modèle cible. Pourtant, l'exécution d'une seule IRT n'est pas suffisante pour produire une alternative architecturale. Dans ce contexte, nous regroupons un ensemble d'IRTs pour former une nouvelle transformation de modèle. Lorsqu'elle est exécutée, cette transformation doit générer une alternative architecturale. En raison de la sémantique d'exécution des langages de transformation de modèle à base de règles [17], les IRTs peuvent être exécutées dans n'importe quel ordre sans modifier le résultat de la transformation. Cette caractéristique est appelée **confluence**. Les instanciations de règles de transformation confluentes sont des IRTs qui peuvent être appliquées en parallèle ou dans n'importe quel ordre pour obtenir le même résultat. En tenant compte de cette définition, nous devons identifier des ensembles d'IRTs confluentes et les regrouper afin de former de nouvelles transformations de modèle.

Le but sous-jacent à l'utilisation de transformations de modèle à base de règles est de gagner du temps et de réduire les erreurs en automatisant les productions et les modifications des alternatives architecturales. Cependant, cette technique ne permet pas d'explorer automatiquement des espaces de conception complexes. Lors de l'étude de systèmes à grande échelle qui contiennent des milliers d'éléments, de nombreuses alternatives de transformation de modèle devraient être développées. L'exploration de ces alternatives devient très complexe et prend beaucoup temps. Pour faire face à cette complexité, nous identifions un ensemble d'instanciations de règles de transformation confluentes et nous les composons afin d'automatiser le développement d'alternatives de transformation de modèle.

9.4.2 Identification automatique d'alternatives architecturales

La composition des transformations de modèles permet de produire de nouvelles transformations de modèles (aussi appelées transformations composites) en combinant des transformations de modèles existantes. La combinaison ici représente la sélection de tout ou partie d'un ensemble d'instanciations de règles de transformation confluentes. Dans cette thèse, nous utilisons des transformations composites afin de: (i) identifier des alternatives architecturales en combinant des instanciations de règles de transformation confluentes, et (ii) générer des alternatives architecturales correctes.

Plus formellement, nous identifions d'abord toutes les instanciations de règles de transformation (IRTs): T = {IRT_{11}, IRT_{12}, IRT_{13}, .., IRT_{21}, .., IRT_{ij}}. Nous regroupons ensuite ces IRTs en fonction de leur confluence: TT = {$IRTC_1$, $IRTC_2$, $IRTC_3$, .., $IRTC_m$}, où chaque $IRTC_i$ représente un ensemble d'IRTs appliqués sur un ensemble (ou tuple) d'éléments du modèle source:

$$IRTC_1 = \{IRT_{11}, IRT_{21}, IRT_{31}\},$$
$$IRTC_2 = \{IRT_{11}, IRT_{21}, IRT_{32}\},$$
$$IRTC_3 = \{IRT_{11}, IRT_{21}, IRT_{33}\},$$
$$....$$

Ensuite, nous composons ces ensembles d'instanciations de règles de transformation confluentes ($IRTC$) pour obtenir toutes les transformations composites possibles (TCs):

$$TC_1 = \{IRTC_1, IRTC_2\},$$
$$TC_2 = \{IRTC_1, IRTC_3\},$$
$$TC_3 = \{IRTC_1, IRTC_2, IRTC_3\},$$
$$....$$

Enfin, nous exécutons les transformations composites produites pour générer l'architecture correspondante.

Pour mieux comprendre la composition des ensembles d'instanciations de règles de transformation confluentes, nous prenons l'exemple suivant: l'allocation de deux composants logiciels (Sw_1, Sw_2) sur deux composants matériels (Hw_1, Hw_2). Nous identifions quatre instanciations de règles de transformation T = {IRT_{11}, IRT_{12},IRT_{21}, IRT_{22}}, où

chaque instanciation de règle de transformation (IRT_{ij}) représente l'allocation d'un composant logiciel i sur un composant matériel j. Par exemple, IRT_{11} représente l'allocation de Sw_1 sur Hw_1, et IRT_{12} représente l'allocation de Sw_1 sur Hw_2.

Ensuite, on regroupe les IRTs en fonction de leur confluence. IRT_{11} et IRT_{12} (respectivement IRT_{21} et IRT_{22}) ne sont pas confluentes: en appliquant IRT_{11} avec IRT_{12}, nous générons un modèle cible où la composante logicielle Sw_1 est allouée en même temps sur deux composants matériels. Ainsi, dans cet exemple, nous avons quatre ensembles d'instanciations de règles de transformation confluentes ($IRTCs$):

$$IRTC_1 = \{IRT_{11}, IRT_{21}\},$$

$$IRTC_2 = \{IRT_{11}, IRT_{22}\},$$

$$IRTC_3 = \{IRT_{12}, IRT_{21}\},$$

$$IRTC_4 = \{IRT_{12}, IRT_{22}\},$$

En composant ces IRTCs, nous produisons nos transformations composites: $TC_1 = \{IRTC_1, IRTC_2\}$, $TC_2 = \{IRTC_1, IRTC_3\}$, $TC_3 = \{IRTC_1, IRTC_2, IRTC_3\}$. Les transformations composites produites représentent des allocations différentes des composants logiciels sur les composants matériels.

Cependant, pas toutes les compositions d'ensembles d'instanciations de règles de transformation confluentes génèrent des transformations composites valides. Une transformation composite valide est une transformation qui, une fois exécutée, produit une alternative architectural correcte.

Dans l'exemple défini ci-dessus, supposons que nous avons la contrainte structurelle suivante: un composant logiciel peut être alloué sur un seul composant matériel. Compte tenu de cette contrainte, la transformation composite TC_1 n'est pas valide. Une fois exécuté TC_1 génère une alternative architectural qui ne respecte pas la contrainte structurelle prédéfinie: le même composant logiciel Sw_1 est alloué sur deux composants matériels différents (Hw_1 et Hw_2) en même temps en appliquant $IRTC_1$ et $IRTC_2$. Dans ce cas la, les ensembles d'instanciations de règles de transformation confluentes ($IRTC_1$ et $IRTC_2$) sont appelés **alternatives de transformation atomiques**. Étant donné un ensemble d'instanciation de règle de transformation confluente $IRTC_1$, $IRTC_2$ est une alternative de transformation atomique si elle contient des instances de règles qui ne sont pas confluentes avec des instances de règles de $IRTC_1$. Dans ce cas, $IRTC_1$ et $IRTC_2$ ne peuvent pas être appliqués sur le même modèle source en parallèle, et soit $IRTC_1$ ou $IRTC_2$ doit être appliquée.

Ainsi, afin de générer des alternatives architecturales correctes, les transformations composites doivent être validées en identifiant des alternatives de transformation atomiques. La validation des transformations composites est décrite dans la sous-section suivante.

9.4.3 Validation des transformation composites

Pour assurer que les transformations composites génèrent des architectures correctes, nous utilisons deux techniques de validation: a-posteriori et a-priori. Plus précisément, nous utilisons une validation a-posteriori pour évaluer l'impact des transformations composites sur un ensemble de PNFs, et une validation a-priori pour construire des transformations composites valides.

9.4.3.1 Validation a-posteriori

La validation a-posteriori consiste à exécuter une transformation composite, à évaluer l'impact du composite sur les PNFs et à vérifier si le modèle cible résultant (i) respecte un ensemble de contraintes structurelles; et (ii) contient des PNFs conformes à certaines exigences définies par les concepteurs logiciels. Bien que la validation a-posteriori présente un grand avantage pour valider des transformations composites, elle présente également des inconvénients. Cette validation prend beaucoup de temps et est très coûteuse: elle nécessite d'exécuter systématiquement chaque transformation composite, d'évaluer le modèle cible résultant par rapport aux PNFs; et de vérifier l'exactitude du modèle cible. Par conséquent, il serait préférable d'utiliser un mécanisme qui valide a-priori les transformations composites.

Dans cette thèse, l'évaluation des PNFs est quantitative. Nous devons donc calculer l'impact des transformations de modèles sur les PNFs. Cet impact est difficile à calculer a priori. Par conséquent, nous utilisons la validation a-posteriori pour évaluer les PNFs. D'autre part, les contraintes structurelles sont qualitatives, et pour ce faire, nous utilisons la validation a-priori. Cette technique de validation est décrite dans les paragraphes suivants.

9.4.3.2 Validation a-priori

La validation a priori consiste à déterminer à la construction de transformations composites si elles génèrent des solutions architecturales correctes. Plus précisément, pour créer des transformations composites valides, nous devons transférer les contraintes structurelles exprimées sur les solutions de conception architecturale en contraintes sur le mécanisme de composition de transformation du modèle.

Les contraintes structurelles sont généralement exprimées à l'aide de langages dédiés tels que le Object Constraint Language (OCL) [59]. OCL est déjà largement utilisé pour exprimer des contraintes sur les modèles. C'est un langage déclaratif qui fournit des règles pour spécifier des contraintes structurelles sur les instances d'un modèle donné. Une fois que les contraintes OCL sont exprimées sur les méta-modèles, nous pouvons vérifier si les modèles respectent ou non les contraintes. Cependant, dans cette thèse, nous visons à exprimer des contraintes structurelles sur des transformations de modèles (développées à l'aide d'un langage spécifique tel que ATL) et non sur des modèles abstraits. Il est difficile d'exprimer des contraintes structurelles de l'OCL sur des alternatives de transformation de modèles. Au lieu de cela, nous adoptons la solution suivante.

Dans cette thèse nous produisons des transformations composites en sélectionnant des instanciations de règles de transformation confluentes (*IRTCs*) à partir de l'ensemble de toutes les instanciations de règles de transformation identifiées T. Pour vérifier les contraintes structurelles lors de la création de transformations composites, nous contraignons la sélection des *IRTCs* en utilisant des formules booléennes. Plus formellement, nous nous basons sur les points suivants:

- T = $\{IRT_{11}, IRT_{12}, .., IRT_{ij}\}$, l'ensemble de toutes les instanciations de règles de transformation,

- IRT_i une instanciations de règles de transformation de l'ensemble T.

- T_i un sous ensemble de l'ensemble T qui ne contient pas IRT_i (T-$\{IRT_i\}$).

- La fonction suivante: $Select: \quad T \quad \rightarrow \mathbb{B} = \{Vrai, Faux\}$
  $IRT \quad \rightarrow b$, ou b est $Vrai$ si IRT doit être sélectionné pour être dans le transformation composite, et $Faux$ si IRT_i doit être exclut de la transformation composite.

- *BoolExpr*: expression booléenne sur des ensembles d'instanciations de règles de transformation. *BoolExpr* Il utilise la fonction (i) Select, et (ii) les opérateurs booléens simples et, ou, et non ($\wedge$, $\vee$, and $\neg$). Par exemple, $BoolExpr_1$ est égal à: $[Select(IRT_{11}) \vee Select(IRT_{12})]$. Supposons dans cet exemple que IRT_{11} alloue un composant logiciel Sw_1 sur un composant matériel Hw_1 et IRT_{12} alloue le même composant logiciel Sw_1 sur un composant matériel différent Hw_2. Dans cet exemple, $BoolExpr1$ définit la propriété de confluence: nous pouvons sélectionner IRT_{11} **ou** IRT_{12} pour l'allocation.

Sur la base de ces points, nous définissons une formule booléenne appelée *Contrainte_Structurelles* qui exprime des contraintes structurelles sur les instanciations de règles de transformation (IRTs):

$$\textbf{Contraintes\_Structurelles} = \bigwedge_{\mathbf{i=1}}^{\mathbf{N}} (\textbf{Select}(\textbf{IRT}_\mathbf{i}) \Rightarrow \textbf{BoolExpr}(\textbf{T}_\mathbf{i})). \tag{9.1}$$

Dans l'équation 9.1, la formule booléenne *Contrainte_Structurelles* exprime pour chaque instanciation de règle de transformation IRT_i les instanciations de règles de transformation qui peuvent être composées avec IRT_i (inclus) et celles qui ne peuvent pas être composées avec IRT_i (exclu).

Par exemple, considérons T un ensemble de quatre instanciations de règles de transformation: T = $\{IRT_{11}, IRT_{12}, IRT_{22}, IRT_{21}\}$. Chaque IRT alloue un composant logiciel sur un composant matériel. En plus de cela, nous avons la contrainte structurelle suivante: différents composants logiciels ne peuvent être alloués sur les mêmes composants matériels. Cette contrainte structurelle peut être exprimée comme une conjonction des expressions booléennes suivantes:

$$\textbf{Contraintes\_Structurelles} =$$
$$Select(IRT_{11}) \Rightarrow [Select(IRT_{22}) \wedge \neg(Select(IRT_{21}) \wedge Select(IRT_{12}))]$$
$$\wedge\, Select(IRT_{12}) \Rightarrow [Select(IRT_{21}) \wedge \neg(Select(IRT_{22}) \wedge Select(IRT_{11}))].$$

Dans cet exemple, la sélection de IRT_{11} implique de sélectionner IRT_{22} et de ne pas sélectionner à la fois IRT_{12} et IRT_{21}. La sélection de IRT_{11} avec IRT_{12} entraîne une transformation composite invalide qui viole la caractéristique de confluence: le composant logiciel Sw_1 est alloué sur deux composants matériels Hw_1 et Hw_2. En plus de cela, en sélectionnant IRT_{11}, nous ne pouvons sélectionner que IRT_{22}.

Une transformation composite valide est ainsi définie par une fonction *Select* qui satisfait l'équation 9.1. Compte tenu de cette fonction, les alternatives de transformation atomiques sont identifiées et appliquées sur le modèle d'entrée. Pour trouver la fonction Select, nous décidons d'utiliser les techniques de résolution SAT [60]. Ces techniques permettent d'évaluer (i) automatiquement l'existence de solutions (alternatives de transformation atomique) à l'équation 9.1, et (ii) d'extraire les solutions trouvées.

Par ailleurs, nous ne définissons pas les contraintes structurelles directement sur les *IRTs*. En d'autres mots nous n'énumérons pas manuellement toutes formules booléennes sur les *IRTs*. Nous automatisons la création des formules booléennes en utilisant un langage interne appelé **TRC (Transformation Rules Catalog)**. Le TRC permet de (i) identi-

fier toutes les règles de transformation à appliquer à un modèle source donné, et (ii) fournir un moyen de formaliser les contraintes structurelles sur l'instanciation d'application des règles de transformation ATL.

Ceci conclut la présentation de notre solution pour résoudre le premier problème abordé dans cette thèse: (i) formaliser les alternatives architecturales en composant des alternatives de transformation atomique; (ii) valider a priori les transformations composites résultantes; et (iii) évaluer a-posteriori les PNFs des transformations composites. Le deuxième problème abordé dans cette thèse est l'exploration des transformations composites et l'identification des alternatives architecturales Pareto-optimal. Nous décrivons cette solution dans la section suivante.

9.5 Exploration de transformations composites

Le deuxième problème abordé dans cette thèse consiste à identifier des alternatives architecturales optimales (ou quasi-optimales). Puisque nous automatisons la modélisation d'alternatives architecturales à l'aide de transformations composites, le processus d'exploration et de sélection est réalisé sur un espace de conception composé de transformations composites.

Lorsque le nombre de transformations composites possibles est très important, il est difficile de toutes les identifier, de les évaluer et de sélectionner les transformations composites optimales (ou presque optimales). Au lieu de cela, nous utilisons des méthodes heuristiques telles que les algorithmes évolutionnistes (AEs) pour converger rapidement vers des transformations composites optimales (ou presque optimales). Cette approche est décrite dans la sous-section suivante.

9.5.1 Exploration de transformations composites

Afin d'explorer un espace de conception de transformations composites, nous définissons un framework d'exploration basé sur les principes d'AEs. La figure 9.1 illustre les quatre étapes du framework proposé.

9.5.1.1 Étape 1

L'AE commence habituellement à partir d'une population initiale de génomes générés de façon aléatoire (transformations composites dans notre cas), où chaque génome est composé d'un ensemble de gènes (alternatives de transformation atomique dans notre cas).

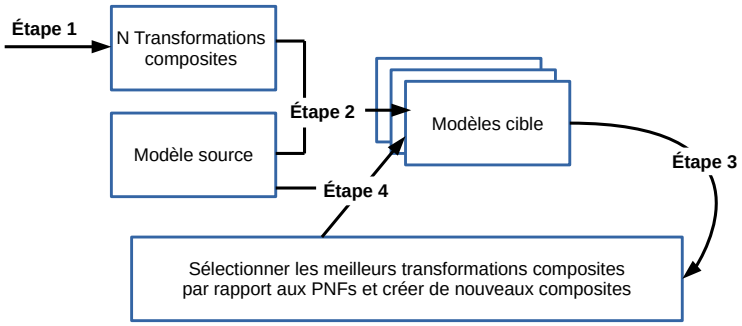

Figure 9.1: Vue abstraite du framework d'exploration

Cette population initiale a une taille fixe de transformations composites N (N peut être fixé, par exemple, par les utilisateurs finaux de notre framework).

9.5.1.2 Étape 2

À partir du modèle source, et de la population initiale de transformations composites, nous produisons N modèles cibles. Plus précisément, nous exécutons chaque transformation composite pour générer le modèle cible correspondant. Les modèles cibles générés sont ensuite analysés par rapport aux PNFs. Les résultats de cette analyse sont par la suite utilisés pour sélectionner les meilleures transformations composites. Les transformations sélectionnées seront utilisées par la suite pour créer de nouvelles transformations composites.

9.5.1.3 Étape 3

Pour sélectionner et créer des transformations composites, nous nous appuyons sur les principes d'AEs: sélection et opérateurs génétiques. Dans notre cas, l'opérateur de sélection consiste à trouver les meilleures transformations composites par rapport aux résultats d'analyses obtenus dans l'étape 2.

Ensuite, les transformations composites sélectionnées sont utilisées pour créer de nouvelles transformations composites. Plus précisément, la création de ces nouvelles transformations composites s'effectue à l'aide des opérateurs de mutation et de croisement (aussi appelés opérateurs génétiques). L'opérateur de croisement consiste à prendre plus d'un génome et à produire un nouveau génome à partir d'eux. De nombreux opérateurs de croisement existent dans la littérature. Par exemple, l'opérateur de croisement à deux points vise à créer de nouveaux génomes à partir de ceux existants en (i) choisissant aléatoirement deux points dans la structure des génomes existants (les gènes); et (ii) échangeant les gènes entre les deux points.

Lors de l'application des opérateurs génétiques sur n'importe quel type de génome, il faut prendre en compte une contrainte importante: les génomes sur lesquels nous appliquons les opérateurs génétiques doivent avoir la même taille. Nous respectons cette contrainte en fixant la taille des transformations composites: toutes les transformations composites ont le même nombre m d'alternatives de transformations atomiques (*ATAs*): $TC_i = \{ATA_j\}_{j=1,..,m}$.

De plus, nous organisons les transformations composites en tableaux. Les tableaux permettent de regrouper un ensemble de variables du même type (alternatives de transformation atomique dans notre cas) auxquelles on peut accéder par un index numérique. Comme on peut le voir sur la figure 9.2, l'indice 3 contient l'alternative de transformation atomique ATA_{31}. Enfin, nous donnons un ordre spécifique pour les alternatives de transformation atomique dans les tableaux: les transformations atomiques sont placées sur le même index dans différentes tableaux. Par exemple, dans la figure 9.2 ATA_{31} et ATA_{32} sont placés sur le même index 3. Ce codage facilite l'application des opérateurs génétiques: le croisement, l'échange d'éléments, et la mutation peuvent être faites en choisissant des aléatoirement les index d'un tableau.

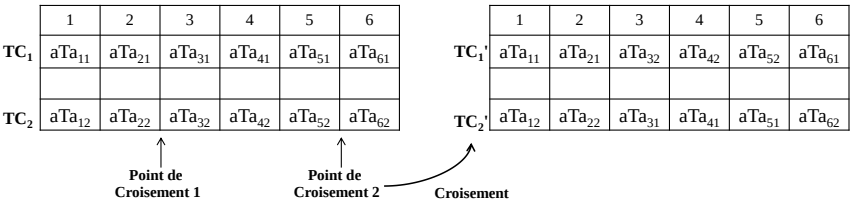

Figure 9.2: L'application du croisement sur des transformations composites

La figure 9.2 illustre l'application de l'opérateur de croisement en deux points sur les transformations composites. Dans cet exemple, lorsque nous appliquons la validation a-priori pour produire les transformations composites initiales, nous identifions les alternatives de transformation atomique: ATA_{i1} et ATA_{i2}. Ensuite, ces alternatives sont placées dans le même index dans les tableaux (transformations composites). Par exemple, ATA_{51} ne peut pas être composé avec ATA_{52}. Ainsi, ATA_{51} et ATA_{52} sont des alternatives et ils sont placés sur le même index dans différents tableaux. En appliquant l'opérateur de croisement, deux transformations composites valides, TC_1 et TC_2, sont coupées entre les deux points (2,3) et (5,6) pour produire deux nouvelles transformations composites TC'_1 et TC'_2. Grâce à notre codage, nous n'avons pas à vérifier la validité de TC'_1 et TC'_2. Puisque nous avons ordonné (avec les index des tableaux) les alternatives de transformation atom-

ique des transformations composites, nous pouvons les échanger et produire de nouvelles transformations composites valides sans répéter le processus de validation.

9.5.1.4 Étape 4

Puisque les AEs sont des processus itératifs, nous itérons sur les étapes 2 et 3 jusqu'à ce qu'un critère de convergence soit atteint. Les critères de convergence communs sont:

1. Un nombre fixe d'itérations atteint.
2. Budget alloué (temps de calcul / argent) atteint
3. Les nouvelles solutions crées ne produisent plus de meilleurs résultats.

Dans notre approche, nous utilisons les premier et troisième critères de convergence pour identifier les transformations composites optimales (ou presque optimales).

Dans cette thèse, nous utilisons des chaînes de transformation de modèles pour formaliser des alternatives architecturales. Ainsi, pour identifier les alternatives architecturales répondant au mieux à un compromis entre des PNFS conflictuelles, nous devons explorer un espace de conception composé de chaînes de transformation de modèles. Cette version étendue de notre approche est décrite dans la sous-section suivante.

9.5.2 Exploration de chaines de transformations de modèles

Pour trouver des alternatives architecturales optimales (ou presque optimales), nous formalisons ces alternatives à l'aide de chaînes de transformation de modèles. Une chaîne de transformation de modèle est composée de différents maillons, ou chaque maillon représente une transformation de modèle une fois exécuté génère un modèle intermédiaire, et le dernier maillon génère le modèle cible.

Comme on peut le voir sur la figure 9.3, le modèle intermédiaire de chaque maillon de la chaîne peut être composé d'un nombre différent d'éléments (tailles différentes). Dans la figure 9.3, en appliquant le maillon T1, nous répliquons un certain nombre de fois les éléments du modèle source. Puisque nous avons différentes alternatives (dupliquer, tripliquer, etc), chaque maillon produit son propre ensemble de modèles intermédiaires. Ces modèles seront utilisées comme entrée du maillon suivant, et produiront un autre ensemble de modèles intermédiaire de tailles différentes. Dans la figure 9.3, en appliquant le maillon T2, nous associons les éléments répliqués et produisons des modèles intermédiaires de tailles différentes. Ainsi, la taille du modèle intermédiaire de chaque maillon dépend de la sortie des maillons précédents de la chaîne.

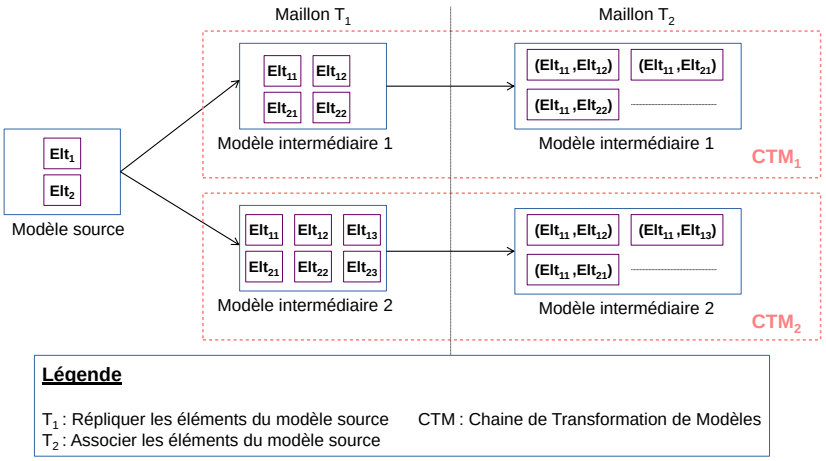

Figure 9.3: Chaines de transformation de modèles

Une fois que nous définissons des chaînes de transformation de modèle, nous devons identifier l'ensemble des solution optimales (ou presque optimales). Pour trouver cet ensemble, nous (i) appliquons les étapes des AES (sélection et opérateurs génétiques) sur ces chaînes, et (ii) sélectionnons les chaînes de transformation modèles optimales (ou presque optimales). Pour développer cette approche, nous comptons sur notre framework d'exploration (défini dans la section précédente). Cependant, des sous-problèmes difficiles doivent être résolus afin d'appliquer notre framework sur les chaînes de transformation de modèles. Dans ce qui suit, nous précisons ces sous-problèmes et donnons nos propositions pour les résoudre:

1. Afin de trouver des chaînes de transformation de modèles optimales (ou presque optimales), nous évaluons chaque maillon de la chaîne en fonction d'un ensemble de PNFs conflictuels. Notre premier sous-problème est que le modèle intermédiaire produit par un maillon de la chaîne ne peut pas être analysé parce que d'autres maillons influent sur le résultat de l'analyse. Dans ce cas, l'exploration ne peut être faite avec des maillons isolés, mais nous devons considérer des alternatives de chaînes de transformations de modèles lors de l'évaluation de l'impact sur les PNFs. Pour résoudre ce premier sous-problème, nous proposons la solution suivante. Pour chaque maillon, nous définissons des alternatives de transformation de modèle qui seront chaînées aux alternatives de transformation des maillons suivants, et nous évaluons uniquement le modèle cible généré par le maillon final.

2. L'espace de conception contient des chaînes de transformation de modèles (génomes) ayant un nombre différent d'alternatives de transformation atomiques (gènes). Les génomes sur lesquels nous appliquons les opérateurs génétiques de l'AE doivent

avoir la même taille (même nombre de gènes). Pour remédier à ce deuxième sous-problème, nous appliquons les opérateurs génétiques sur les alternatives de transformation de modèle enchaîné, au lieu de les appliquer aux alternatives de transformation atomique d'une transformation de modèle donnée.

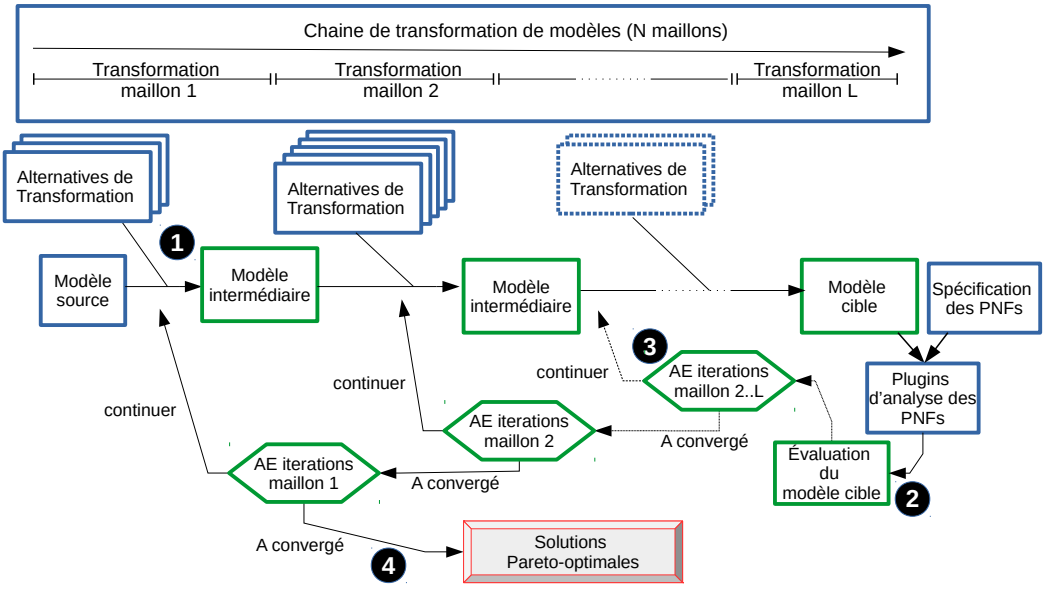

Figure 9.4: Exploration de chaines de transformations de modèles

La figure 9.4 donne un aperçu du processus que nous proposons pour explorer une chaîne de transformations de modèle: de la définition d'une chaîne, ayant pour chaque maillon un ensemble d'alternatives de transformations, nous produisons d'abord un modèle intermédiaire qui résulte de la composition de ces alternatives. Cette première étape est mise en évidence avec la puce 1 sur la figure 9.4 et répétée pour chaque maillon de la chaîne de transformation jusqu'à ce que le modèle cible soit produit.

Nous supposons que les modèles intermédiaires ne contiennent pas toutes les informations nécessaires pour analyser les PNFs des alternatives architecturales: les modèles intermédiaires sont enrichis dans chaque maillon de la chaîne de transformation. Une fois produit, le modèle cible est analysé par rapport aux PNFs (voir la puce 2 sur la figure 9.4) et les résultats de l'analyse sont utilisés pour évaluer les transformations composites.

Le processus que nous proposons est itératif: chaque itération produit, exécute et évalue une sous-chaîne de transformations composites. De plus, en raison de la complexité combinatoire de l'exploration de l'espace de conception, il n'est pas possible d'énumérer, d'exécuter et d'évaluer toutes les transformations composites. En conséquence, nous nous appuyons sur des algorithmes évolutionnistes (AEs) pour mettre en œuvre cette explo-

ration itérative (voir la puce 3 sur la figure 9.4). De plus, on peut voir sur la figure 9.4 que le processus proposé est constitué de boucles intégrées, chaque boucle étant dédiée à explorer des transformations composites d'un maillon donné dans la chaîne de transformation. Lorsqu'une boucle intérieure a convergé, d'autres candidats à la transformation peuvent être évalués pour la boucle externe, produisant ainsi un nouveau modèle intermédiaire pour la boucle interne. Les critères de convergence pour chaque boucle reposent sur des critères de convergence des AEs et sont paramétrés par l'utilisateur final de notre approche. Lorsque la boucle de niveau supérieur a convergé, le processus d'exploration s'arrête et renvoie l'ensemble des modèles optimales (ou presque optimales) qui ont été trouvés (voir la puce 4 de la figure 9.4).

Ceci conclut la présentation des contributions techniques de cette thèse. Nous passons maintenant à l'évaluation expérimentale de nos propositions dans la section suivante.

9.6 Expérimentations et évaluation de l'approche

Après avoir présenté nos contributions dans les sections précédentes, nous présentons maintenant les activités de validation que nous avons menées pour évaluer nos propositions. Afin d'évaluer et de valider notre approche, nous proposons de (i) formuler un ensemble de questions de recherche visant à évaluer notre approche par rapport à certains critères, (ii) appliquer notre approche à des cas d'études différents, et (iii) répondre aux questions de recherche formulées en utilisant les résultats expérimentaux.

Dans cette thèse, nous développons un framework qui explore un espace de conception d'alternatives architecturales en composant des transformations de modèle et en appliquant des algorithmes évolutionnistes sur des transformations composites crées. Afin d'évaluer la qualité de notre approche, nous proposons de répondre aux questions de recherche suivantes:

QR1: notre approche peut-elle être réutilisée pour résoudre différents types de problèmes d'optimisation?

QR2: Est-il plus efficace de procéder à la validation des contraintes structurelles a priori au lieu d'a posteriori?

QR3: Quelle est la distance entre les résultats obtenus avec notre approche et d'autres approches telles que la programmation linéaire ou les algorithmes d'exploration aléatoire?

QR4: Quelle quantité de ressources (mémoire et temps de calcul) notre approche requiert-elle pour explorer un grand espace de conception de transformations composites?

Afin de répondre à ces questions, nous proposons d'abord de définir des cas d'études expérimentaux. Ensuite, nous appliquerons notre framework à ces cas d'études. Enfin, sur la base des résultats expérimentaux, nous évaluons notre framework.

9.6.1 Évaluation de la généricité du framework

Pour répondre à la première question de recherche (QR1), c'est-à-dire la généricité de notre framework, nous (i) définissons deux cas d'études différents (ayant des problèmes d'optimisation différents), (ii) appliquons notre framework aux cas d'études, et (iii) vérifions si notre framework identifie des alternatives architecturales optimales (ou presque optimales).

9.6.1.1 Modèle du premier cas d'étude

L'architecture système considéré dans le premier cas d'étude est une architecture logicielle représentant un système de contrôle de train. Cette architecture logicielle se compose de deux processus interconnectés: le processus Automatic_Train_Operation (ATO), chargé de contrôler la position, la vitesse et l'accélération du train. L'autre processus, appelé Automatic_Train_Protection (ATP), communique avec l'ATO afin de vérifier la validité des données calculées par l'ATO. La figure 9.5 illustre l'architecture AADL du processus Automatic_Train_Operation (ATO). Ce processus est constitué de quatre threads interconnectés (par des ports) visant à contrôler la position, la vitesse et l'accélération du train. De plus, l'ATO et l'ATP sont reliés par des connections AADL entre leurs ports (voir les ports sur la partie droite de la figure figure 9.5).

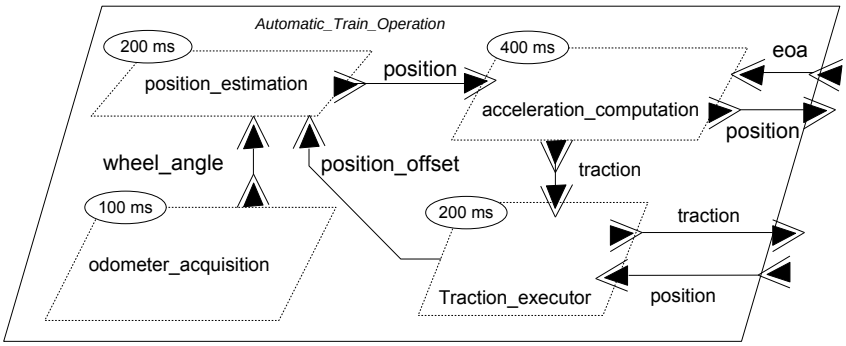

Figure 9.5: Modèle du premier cas d'étude

9.6.1.2 Alternatives de transformations de modèles du premier cas d'étude

Sur le modèle présenté dans la sous-section précédente, nous appliquons une transformation de modèle qui implémente différentes communications entre les threads. Cette transformation est composée de 120 règles ATL et fournit trois alternatives: LowET (Low Execution Time), LowMF (Low Memory Footprint) et LowMC (Low Maintenance Cost) qui représentent les solutions les plus efficaces en termes de temps d'exécution, d'empreinte mémoire et de coût de maintenance. Chaque alternative de transformation transforme les ports AADL et les connexions entre les threads d'un même processus.

LowET et LowMF utilisent la même fonction mathématique permettant d'implémenter les communications entre threads. Donc, ces deux alternatives sont compatibles et peuvent être composées pour créer de nouvelles transformations. Cependant, pour maintenir ces solutions il faudrait une connaissance accrue des protocoles de communication. De son coté, L'alternative LowMC est basée sur une liste chainée et des sémaphores pour implementer les communications entre threads. Cette solution est meilleure pour la maintenance. Cependant, elle est moins efficace en terme de temps de réponse et empreinte mémoire car elle utilise des sémaphores. En plus de cela, il est impossible de composer LowMC avec (LowET et LowMF) car les solutions sont basées sur des principes différents.

9.6.2 Résultats expérimentaux pour le premier cas d'étude

Pour les expérimentations nous utilisons le modèle architecturale et les transformations de modèle présentées dans les sous-sections précédentes. Nous supposons que les modèles source et cible contiennent des propriétés AADL pour évaluer les PNFs: consommation de ressources CPU, empreinte mémoire et maintenabilité. De plus, nous formalisons les contraintes structurelles entre les règles de transformation de modèles. A partir de ces artefacts d'entrée, notre approche identifie les instanciations des règles de transformation (IRTs), détecte les IRTs confluents et produit des formules booléennes pour créer des alternatives de transformation atomique (ATAs).

La figure 9.6 montre que notre approche a permis d'identifier 5 solutions presque optimales. En effet, dans cette figure on peut observer que les solutions du groupe 1 ont une meilleure marge en termes de temps d'exécution, mais une marge plus faible en empreinte mémoire et maintenabilité. Les solutions du groupe 2 ont une meilleure marge en terme d'empreinte mémoire (mais une plus faible marge en termes de maintenabilité et de temps d'exécution). Enfin, les solutions du groupe 3 ont une meilleure marge en terme de maintenabilité (mais une plus faible en terme de temps d'exécution et de mémoire). Notons

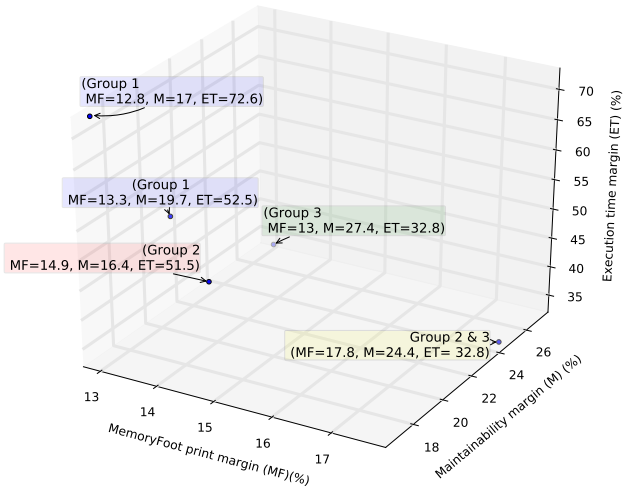

Figure 9.6: Résultats expérimentaux du premier cas d'étude

que certaines solutions appartiennent à deux groupes car elles ont de bonnes performances pour deux PNFs, mais conduisent à une pire solution par rapport au troisième PNF.

9.6.2.1 Modèle du deuxième cas d'étude

Dans le deuxième cas d'étude, les concepteurs ferroviaires veulent organiser toutes les ressources opérationnelles afin que leur système atteigne les PNFs suivants: sûreté, fiabilité, disponibilité et temps de réponse. Afin de réaliser cela, les concepteurs utilisent deux patrons de conception différents. Premièrement, ils appliquent des patrons de réplication permettant de reproduire des composants logiciels afin d'accroître la fiabilité et/ou la sûreté du système. Deuxièmement, les concepteurs allouent les composants logiciels (répliqués) sur les composants matériels.

9.6.2.2 Alternatives de transformations de modèles du deuxième cas d'étude

Les transformations de modèle utilisées pour le deuxième cas d'étude mettent en œuvre un patron de conception classique dans les systèmes embarqués critiques: la réplication des composants logiciels et leur allocation sur les composants matériels.

Réplication des composants logiciels: nous considérons deux transformations de modèle pour la réplication des composants logiciels, à savoir: Deux parmi Trois (2/3) et Deux fois Deux parmi 2 (2*2/2). Lors de l'application de la transformation 2/3, chaque composant logiciel est répliqué deux fois (le résultat est de trois répliques: le processus plus ses deux répliques), puis toutes les répliques fonctionnent en parallèle. Les commu-

nications entre les répliquent sont également répliquées de sorte que chaque processus et chacune de ses répliques communiquent avec leurs processus successeurs et toutes leurs répliques. Dans le cas de l'application de la transformation 2*2/2, chaque processus est répliqué trois fois (c'est-à-dire quatre répliques.

Allocation des composants logiciels: consiste à allouer les composants logiciels sur les composants matériels: chaque composant logiciel doit être affecté à un processeur pour exécuter ses threads.

9.6.3 Résultats expérimentaux

Après avoir défini tous les artefacts d'entrée (le modèle, la chaîne de transformation du modèle), nous configurons notre approche pour (i) identifier les IRTs confluents de la chaîne de transformation du modèle, (ii) créer les ATAs, et (ii) identifier les alternatives architecturales optimales (ou presque optimales).

Pour nos expérimentations, nous avons utilisé un modèle source composé de 3 composants logiciels et 5 composants matériels multi-cœurs. En appliquant les transformations de réplication (c'est-à-dire 2/3 ou 2*2/2), l'architecture logicielle est constituée de 9 à 12 composants logiciels. La plate-forme matérielle est suffisamment sophistiquée pour fournir un ensemble important d'allocations possibles: 20 pour chaque processus répliqué. Ainsi, cette expérience fournit un espace de conception très important.

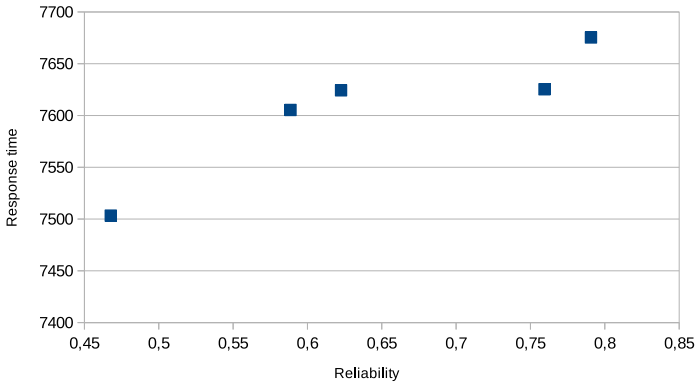

Figure 9.7: Résultats expérimentaux du second cas d'étude

La figure 9.7 montre les résultats de l'application de notre approche sur le modèle présenté auparavant. Plus précisement, en appliquant notre approche nous avons identifié 5 solutions presque optimales (en terme de fiabilité et de temps de réponse). Ainsi, le concepteur peut choisir la solution qui lui convient le mieux.

Pour résumer, nous avons appliqué notre framework sur deux problèmes d'optimisation différents (raffinement de modèle, et réplication/allocation de composants logiciels), et à partir des résultats expérimentaux nous concluons que notre approche peut être appliquée

pour résoudre différents problèmes d'optimisation. Ces résultats démontrent la généricité de notre framework.

9.6.4 Évaluation de la qualité du framework

Afin d'évaluer la qualité de notre approche, nous proposons de répondre aux questions de recherche restantes.

9.6.4.1 Validation a-priori et a-posteriori

Méthode d'évaluation: Nous comparons le temps de convergence requis par notre approche avec le temps de convergence requis par une approche basée sur les AEs mais en utilisant une validation a-posteriori des contraintes structurelles.

Table 9.1: Comparaison des techniques de validation

Technique de validation	Nombre d'itérations	Nombre de solutions optimales	Temps de convergence
A posteriori	2000	0	Pas convergé
A priori	2000	3	a convergé après 11 heures

Résultats: Le tableau 9.1 présente les résultats de l'exécution de notre approche et celle basé sur une validation a-posteriori sur le modèle source présenté dans la sous-section 9.6.3. Sur deux mille architectures générées par l'approche basée sur une validation a posteriori, aucune n'est correcte en ce qui concerne les contraintes structurelles. Ainsi, cette approche ne converge pas et aucune solution n'est trouvée. D'autre part, l'approche basée sur une validation a-priori est capable de converger et d'identifier 3 architectures presque optimales en évaluant plus de deux mille architectures correctes. Cela montre l'importance de valider a priori les transformations composites plutôt qu'a-posteriori.

9.6.4.2 Comparaison avec d'autres algorithmes d'optimisation

Méthode d'évaluation: Nous comparons la distance entre les solutions obtenues avec notre approche et un optimum local théorique obtenues en utilisant la programmation linéaire ou un algorithme d'optimisation aléatoire. Les techniques de programmation linéaire sont connues pour être très efficaces. C'est donc une très bonne méthode de référence. Nous utilisons donc cette méthode pour calculer l'optimum local correspondant au meilleur optimum local et le pire optimum local: le meilleur optimum local est obtenu en maximisant une fonction objective tandis que le pire est obtenu en minimisant la fonction objective. Cependant utiliser la programmation linéaire nécessite de linéariser

le problème multiobjectif en une seule fonction objective ce qui est difficile à faire en général.

Résultats: En faisant la comparaison nous avons tiré les conclusions suivantes: les solutions trouvées en utilisant la programmation linéaire ne dominent pas celles trouvées par notre approche. Les solutions trouvées en utilisant notre approche sont bien meilleurs que celles obtenues en utilisant un algorithme aléatoire.

9.6.4.3 Consommation de ressources

Méthode d'évaluation: Pour évaluer la quantité de ressources nécessaires à l'exécution de notre approche, nous mesurons les pics d'espace mémoire et de puissance de calcul qu'il utilisait au moment de l'exécution, ainsi que le temps nécessaire à la convergence.

Table 9.2: Consommation de resources

Modèle source	Temps de convergence	Architectures évalués	Utilisation mémoire
Modèle 1	11h09m	1920	13 GB
Modèle 2	9h39m	2121	12.4 GB
Modèle 3	13h24m	1710	14 GB
Modèle 4	14h23m	1415	13 GB

Résultats: Le tableau 9.2 présente la consommation de ressources mesurée, au moment de l'exécution, pour différents modèles sources ayant différents niveaux d'abstractions. Les expériences ont été faites sur un machine dotée de 22 cœurs. Ces résultats montrent une consommation de ressources élevée dans les différents modèles. En mesurant le temps passé dans différentes parties de notre implémentation, nous avons pu identifier qu'une partie importante du temps de calcul (plus de 50%) est dépensée pour la création des transformations composites. Par conséquent, nous croyons que le temps d'exécution que nous mesurons serait considérablement amélioré en optimisant cette partie de notre approche.

9.7 Conclusions

9.7.1 Conclusion

Dans cette thèse, nous avons décrit une approche permettant d'explorer un espace de conception d'alternatives architecturales, et de sélectionner les alternatives répondant au mieux à un compromis entre des propriétés non fonctionnelles conflictuelles. L'exploration

de tels espaces de conception présente plusieurs problèmes. Certains systèmes complexes peuvent admettre des millions de composants et d'interactions entre ces composants. L'énumération et la modélisation de toutes les alternatives architecturales possibles sont sujettes à des erreur. Par conséquent, il est nécessaire de fournir une solution automatisée permettant d'identifier et de modéliser l'espace de conception des alternatives architecturales. En plus de cela, pas toutes les alternatives architecturales sont correctes par rapport à différentes contraintes structurelles, et les concepteurs doivent vérifier l'exactitude de chaque alternative avant d'explorer l'espace. L'exécution de cette opération est coûteuse et laborieuse, et dans certains cas, les concepteurs peuvent ne pas trouver d'alternative architecturales optimales (ou presque optimales). Enfin, pour explorer un espace de conception d'alternatives architecturales impliquant de multiples propriétés non fonctionnelles conflictuelles, les techniques d'optimisation multiples-objectifs (TOMOs) sont d'un grand intérêt. Cependant, les TOMOs sont basés sur des heuristiques telles que les opérateurs de variation. Ces opérateurs sont généralement appliqués sur des solutions formalisées à l'aide d'un codage binaire ou réel. Ainsi, pour appliquer les TOMO sur un espace de conception d'alternatives architecturales, un encodage spécifique des alternatives est nécessaire.

Pour résoudre ces problèmes, nous proposons un framework d'exploration générique qui combine des techniques d'optimisation multi-objectifs et des techniques de transformation de modèles. Les principales contributions de cette thèse sont résumées dans les paragraphes suivants.

La première contribution de cette thèse consiste à automatiser la formalisation et l'analyse des alternatives architecturales. Dans notre approche, les alternatives architecturales sont générées automatiquement à l'aide d'un langage de transformation de modèle à base de règles.

Nous considérons ce type de transformations de modèles car elles peuvent être réutilisées pour différents problèmes d'optimisation multi-objectifs sans modifier le framework d'exploration. Ainsi, notre framework d'exploration est générique et peut être appliqué pour résoudre différents problèmes d'optimisation. De plus, on peut vérifier a priori (avant l'application d'une transformation composite) l'exactitude des alternatives architecturales concernant les contraintes structurelles en utilisant des formules booléennes et des techniques de résolution SAT. Cela permet d'exclure les transformations composites produisant des architectures incorrectes, et donc de réduire l'espace de conception à explorer. Enfin, des problèmes d'optimisation complexes peuvent être formulés et explorés en chaînant deux ou plusieurs transformations de modèles basées sur des règles. Cependant,

l'espace de conception croît rapidement avec le nombre de transformations composites, ce qui conduit à un problème combinatoire.

Face aux problèmes posés par l'exploration d'alternatives architecturales, et en particulier l'explosion combinatoire des transformations composites et le caractère conflictuel des PNFs, nous avons automatisé la production de transformations composites à l'aide d'une technique bien connue d'optimisation multiobjective: les Algorithmes Évolutionnistes (AEs). Dans cette thèse, les AEs prennent comme entrée une population initiale de transformations composites créées aléatoirement, et appliquent différentes étapes (évaluation, sélection et opérateurs génétiques) pour créer de nouvelles transformations composites. Ces étapes sont répétées un certain nombre de fois jusqu'à produire un ensemble de transformations composites qui génèrent des alternatives architecturales optimales (ou presque optimales).

Afin de considérer des problèmes d'optimisation plus complexes et de faciliter la maintenabilité et la réutilisation des transformations de modèle, nous avons proposé de chaîner deux ou plusieurs transformations de modèle basées sur des règles (le modèle de sortie d'une transformation de modèle devient le modèle d'entrée d'une nouvelle transformation de modèle). Ainsi, nous générons des alternatives architecturales en exécutant des chaînes de transformation modèles. Par conséquent, pour identifier des alternatives optimales (ou presque optimales), nous devons explorer des espaces de conception composés de chaînes de transformation modèles. Pour atteindre cet objectif, nous avons appliqué des AEs sur chaque transformation de modèle (également appelée maillon) d'une chaîne donnée. Plus précisément, nous avons itéré sur les maillons d'une chaîne et produit les transformations composites les plus appropriées pour chaque maillon.

Pour valider l'applicabilité et l'efficacité de notre approche, nous l'avons exécuté sur deux cas d'études différents; et nous avons vérifié les résultats expérimentaux en fonction de différents critères. Ces critères visent à évaluer (i) la généricité de notre approche, (ii) la consommation de ressources (mémoire et temps de calcul) de notre approche, (iii) l'efficacité de notre framework d'exploration par rapport à d'autres solutions d'exploration possibles telles que la programmation linéaire. Tous les résultats expérimentaux obtenus valident la capacité de notre approche à explorer un espace de conception de transformations composites (et des chaînes de transformation de modèles) et d'identifier un ensemble d'alternatives architecturales optimales (ou presque optimales).

9.7.2 Perspectives

La validation de chacune de nos contributions a révélé certains points ouverts pour lesquels nous avons identifié des solutions prometteuses à étudier: La détermination des paramètres à utiliser pour un problème d'optimisation donné est essentielle au succès des algorithmes évolutionnistes. L'utilisation de bons paramètres aide à identifier plus rapidement des solutions optimales. Dans certains travaux [29; 30; 28; 34], les auteurs ont proposé de faire varier dynamiquement les paramètres des algorithmes évolutionnistes. Par exemple, la probabilité d'appliquer le croisement (et la mutation) augmente (ou diminue) en fonction des résultats obtenus dans l'application précédente du croisement (et de la mutation). D'après leurs résultats, les auteurs ont déclaré que la variation des paramètres d'un algorithme évolutionniste permet d'identifier plus rapidement des solutions optimales. Nous prévoyons donc d'intégrer ces travaux dans notre framework d'exploration afin d'évaluer la sensibilité de notre approche par rapport aux variations des paramètres des AEs, et ainsi converger plus rapidement dans des alternatives d'architecture optimales (ou presque optimales).

Dans nos expériences, nous avons considéré des chaînes de transformation de modèles composées de deux transformations de modèles à base de règles: la réplication de transformations de modèles, et l'allocation de composants logiciels. Nous envisageons d'augmenter le champ d'application de notre framework d'exploration en considérant des chaînes de transformation de modèles plus complexes composées de plus de deux transformations de modèles.

Lors de l'application de notre approche sur des cas d'études différents, nous avons observé que notre framework a consacré beaucoup de temps (plus de 50% du temps de calcul) à produire les transformations composites. Par conséquent, nous prévoyons d'améliorer les performances de notre approche afin de pouvoir considérer des cas d'études plus complexes. En particulier, nous prévoyons de mettre en œuvre une runtime ATL qui nous permettra de réduire le temps de création de transformations composites.

10 Publications

1. **E. Borde (Telecom ParisTech), S. Rahmoun (IRT SystemX and Telecom ParisTech), F. Cadoret (Telecom ParisTech), L. Pautet (Telecom ParisTech), F. Singhoff (University of Brest), P. Dissaux (Ellidiss)**. Architecture models refinement for fine grain timing analysis of embedded systems. In *RSP*, New Delhi, India, 2014.

2. **Rahmoun, Smail and Borde, Etienne and Pautet, Laurent**. Automatic selection and composition of model transformations alternatives using evolutionary algorithms. In *1st international ACM Sigsoft Workshop on Variability for Qualities in Software Architecture (VAQUITA'15), associated with the European Conference on Software Architecture, ECSA'15*, Dubrovnik, Croatia, 2015. ACM.

3. **Rahmoun, Smail and Borde, Etienne and Pautet, Laurent**. Multi-objectives Refinement of AADL Models for the Synthesis Embedded Systems (mu-RAMSES). In *Proceedings of the 2015 20th International Conference on Engineering of Complex Computer Systems, ICECCS'15*, Gold Coast, Australia, 2015. IEEE.

4. **Rahmoun, Smail and Mehiaoui-Hamitou, Asma and Borde, Etienne and Pautet, Laurent and Soubiran, Elie**. Multi-Objective Exploration of Architectural Designs by Composition of Model Transformations. In *International Journal on Software and Systems Modeling (SoSyM)*, 2016. Springer.

Bibliography

[1] S. Lianshan and W. Jinyu, *Modeling Nonfunctional Requirements in Software Product Line*, pp. 745–753. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[2] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[3] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, pp. 42–45, Sept. 2003.

[4] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake, "Measuring non-functional properties in software product line for product derivation," in *15th Asia-Pacific Software Engineering Conference (APSEC 2008), 3-5 December 2008, Beijing, China*, pp. 187–194, 2008.

[5] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, pp. 621–645, July 2006.

[6] J.-M. Jézéquel, O. Barais, and F. Fleurey, "Model driven language engineering with kermeta," in *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, (Berlin, Heidelberg), pp. 201–221, Springer-Verlag, 2011.

[7] A. Agrawal, *A Formal Graph Transformation Based Language for Model-to-model Transformations*. PhD thesis, Nashville, TN, USA, 2004. AAI3143620.

[8] F. Jouault and I. Kurtev, "Transforming models with atl," in *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, (Berlin, Heidelberg), pp. 128–138, Springer-Verlag, 2006.

[9] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.

[10] F. Telecom, "Smartqvt: An open source model transformation tool implementing the mof 2.0 qvt-operational language," 2007.

[11] A. Königs and A. Schürr, "Tool integration with triple graph grammars - a survey," *Electron. Notes Theor. Comput. Sci.*, vol. 148, pp. 113–150, Feb. 2006.

[12] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *The Epsilon Transformation Language*, pp. 46–60. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[13] K. Lano, "The uml-rsds manual," 2012.

[14] E. Borde, S. Rahmoun, F. Cadoret, L. Pautet, F. Singhoff, and P. Dissaux, "Architecture models refinement for fine grain timing analysis of embedded systems," in *25th IEEE International Symposium on Rapid System Prototyping (RSP), 2014*, pp. 44–50, Oct 2014.

[15] A. Etien, V. Aranega, X. Blanc, and R. F. Paige, "Chaining model transformations," in *Proceedings of the First Workshop on the Analysis of Model Transformations*, (New York, USA), pp. 9–14, ACM, 2012.

[16] C. Castellanos, E. Borde, L. Pautet, G. Sébastien, and T. Vergnaud, "Improving reusability of model transformations by automating their composition," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 267–274, Aug 2015.

[17] D. Wagelaar, *Composition Techniques for Rule-Based Model Transformation Languages*, pp. 152–167. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[18] J. Sánchez Cuadrado and J. García Molina, "Approaches for model transformation reuse: Factorization and composition," in *1st International Conference on Model Transformations (ICMT 08)*, (Berlin, Heidelberg), pp. 168–182, Springer-Verlag, 2008.

[19] D. Wagelaar, R. Van Der Straeten, and D. Deridder, "Module superimposition: a composition technique for rule-based model transformation languages," *Software & Systems Modeling*, vol. 9, no. 3, pp. 285–309, 2009.

[20] C. A. Coello Coello, C. Dhaenens, and L. Jourdan, *Multi-Objective Combinatorial Optimization: Problematic and Context*, pp. 1–21. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[21] I. P. Stanimirovic, M. L. Zlatanovic, and M. D. Petkovic, "On the linear weighted sum method for multi-objective optimization," *Facta Acta Universitatis*, vol. 26, pp. 49–63, 2011.

[22] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich, "Efficient symbolic multi-objective design space exploration," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, (Los Alamitos, CA, USA), pp. 691–696, IEEE Computer Society Press, 2008.

[23] J. Z. Li, J. Chinneck, M. Woodside, and M. Litoiu, "Fast scalable optimization to configure service systems having cost and quality of service constraints," in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, (New York, NY, USA), pp. 159–168, ACM, 2009.

[24] L. Zeng, B. Benatallah, P. Nguyen, and A. H. H. Ngu, "Agflow: Agent-based cross-enterprise workflow management system," in *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, (San Francisco, CA, USA), pp. 697–698, Morgan Kaufmann Publishers Inc., 2001.

[25] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *Des. Autom. Embedded Syst.*, vol. 2, pp. 165–193, Mar. 1997.

[26] D. P. Solomatine, "Genetic and other global optimization algorithms – comparison and use in calibration problems," in *PROC., 3RD INT. CONF. ON HYDROINFORMATICS, BALKEMA*, pp. 1021–1027, 1998.

[27] M. A. El-Beltagy and A. J. Keane, "A comparison of genetic algorithms with various optimization methods for multi-level problems," *ENG. APPL. OF ARTIFICIAL INTELLIGENCE*, vol. 12, pp. 639–654, 1999.

[28] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '09, (Washington, DC, USA), pp. 61–71, IEEE Computer Society, 2009.

[29] A. Koziolek, H. Koziolek, and R. Reussner, "Peropteryx: Automated application of tactics in multi-objective software architecture optimization," in *Proceedings of the ACM SIGSOFT Conference Quality of Software Architectures*, (New York, NY, USA), ACM, 2011.

[30] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron, "An evolutionary multiobjective optimization approach to component-based software architecture design," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pp. 432–439, June 2011.

[31] L. Grunske, "Identifying "good" architectural design alternatives with multiobjective optimization strategies," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), pp. 849–852, ACM, 2006.

[32] I. Meedeniya and L. Grunske, "An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 229–238, Nov 2010.

[33] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st ed., 2012.

[34] J. Fredriksson, K. Sandström, and M. Åkerholm, *Optimizing Resource Usage in Component-Based Real-Time Systems*, pp. 49–65. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[35] D. K. Friesen and M. A. Langston, "Variable sized bin packing," *SIAM J. Comput.*, vol. 15, pp. 222–230, Feb. 1986.

[36] A. Martens and H. Koziolek, "Automatic, model-based software performance improvement for component-based software designs," *Electron. Notes Theor. Comput. Sci.*, vol. 253, pp. 77–93, Oct. 2009.

[37] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, (New York, NY, USA), pp. 105–116, ACM, 2010.

[38] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, pp. 3–22, Jan. 2009.

[39] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Trans. Evol. Comp*, vol. 6, pp. 182–197, Apr. 2002.

[40] E. Zitzler, K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, T. F. (eds, E. Z. Ler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," 2002.

[41] N. Beume, B. Naujoks, and M. Emmerich, "SMS-EMOA: Multiobjective selection based on dominated hypervolume," *European Journal of Operational Research*, vol. 181, pp. 1653–1669, September 2007.

[42] E. Insfran, J. Gonzalez-Huerta, and S. Abrahão, "Design guidelines for the development of quality-driven model transformations," in *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*, MODELS'10, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2010.

[43] R. Li, M. R. Chaudron, and R. C. Ladan, "Towards automated software architectures design using model transformations and evolutionary algorithms," in *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '10, (New York, NY, USA), pp. 2097–2098, ACM, 2010.

[44] M. L. Drago, C. Ghezzi, and R. Mirandola, "Towards quality driven exploration of model transformation spaces," in *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, (Berlin, Heidelberg), Springer-Verlag, 2011.

[45] B. Schätz, F. Hölzl, and T. Lundkvist, "Design-space exploration through constraint-based model-transformation," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pp. 173–182, March 2010.

[46] A. Ciancone, M. L. Drago, A. Filieri, V. Grassi, H. Koziolek, and R. Mirandola, "The klapersuite framework for model-driven reliability analysis of component-based systems," *Software & Systems Modeling*, vol. 13, no. 4, pp. 1269–1290, 2014.

[47] M. L. Drago, C. Ghezzi, and R. Mirandola, "A quality driven extension to the qvt-relations transformation language," *Comput. Sci.*, vol. 30, pp. 1–20, Feb. 2015.

[48] A. Kavimandan and A. Gokhale, "Applying model transformations to optimizing real-time qos configurations in dre systems," in *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, (Berlin, Heidelberg), pp. 18–35, Springer-Verlag, 2009.

[49] J. Xu, "Rule-based automatic software performance diagnosis and improvement," *Perform. Eval.*, vol. 67, pp. 585–611, Aug. 2010.

[50] J. Denil, M. Jukss, C. Verbrugge, and H. Vangheluwe, *Search-Based Model Optimization Using Model Transformations*, pp. 80–95. Cham: Springer International Publishing, 2014.

[51] Á. Hegedüs, Á. Horváth, and D. Varró, "A model-driven framework for guided design space exploration," *Automated Software Engineering*, vol. 22, no. 3, pp. 399–436, 2015.

[52] G. Loniewski, E. Borde, D. Blouin, and E. Insfran, "An automated approach for architectural model transformations," in *22nd International Conference on Information Systems Development (ISD2013)*, (Sevilla Spain), Sept. 2013.

[53] F. Maswar, M. R. Chaudron, I. Radovanovic, and E. Bondarev, "Improving architectural quality properties through model transformation," in *Proceedings of the 2007 International Conference on Software Engineering Research and Practice*, SERP 2007, pp. 687–693, 2007.

[54] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, ch. Towards a General Composition Semantics for Rule-Based Model Transformation, pp. 623–637. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[55] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 39, pp. 658–683, May 2013.

[56] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, (Berlin, Heidelberg), Springer-Verlag, 2009.

[57] F. P. Basso, R. M. Pillat, T. C. de Oliveira, and L. B. Becker, "Supporting large scale model transformation reuse," in *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, 2013.

[58] R. Heckel, J. Kuster, and G. Taentzer, "Confluence of typed attributed graph transformation systems," in *Graph Transformation* (A. Corradini, H. Ehrig, H.-J. Kreowski,

and G. Rozenberg, eds.), vol. 2505 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2002.

[59] O. M. G. OMG, "Flow latency analysis with the architecture analysis and design language (aadl)." http://www.omg.org/spec/OCL/2.4, 2012.

[60] L. Zhang and S. Malik, "The quest for efficient boolean satisfiability solvers," in *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, (London, UK, UK), pp. 17–36, Springer-Verlag, 2002.

[61] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[62] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[63] D. Le Berre and A. Parrain, "The sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, 2010.

[64] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," MODELS'11, (Berlin, Heidelberg), pp. 623–637, Springer-Verlag, 2011.

[65] N. Srinivas and K. Deb, "Multiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, 1994.

[66] A. Jaouën, E. Borde, L. Pautet, and T. Robert, *PDP 4PS : Periodic-Delayed Protocol for Partitioned Systems*, pp. 149–165. Cham: Springer International Publishing, 2014.

[67] F. Cadoret, T. Robert, E. Borde, L. Pautet, and F. Singhoff, "Deterministic implementation of periodic-delayed communications and experimentation in aadl," in *IEEE 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013*, pp. 1–8, June 2013.

[68] W. Zheng, Q. Zhu, M. D. Natale, and A. S. Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pp. 161–170, IEEE, 2007.

[69] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175–1185, 1990.

[70] K. Tindell and A. Burns, "Guaranteeing message latencies on control area network (can)," in *Proceedings of the 1st International CAN Conference*, Citeseer, 1994.

[71] Q. Zhu, Y. Yang, E. Scholte, M. D. Natale, and A. Sangiovanni-Vincentelli, "Optimizing extensibility in hard real-time distributed systems," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pp. 275–284, IEEE, 2009.

[72] F. Nadi and A. T. Khader, "A parameter-less genetic algorithm with customized crossover and mutation operators," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, (New York, NY, USA), pp. 901–908, ACM, 2011.

[73] S. Meyer-Nieberg and H.-G. Beyer, *Self-Adaptation in Evolutionary Algorithms*, pp. 47–75. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.