Noname manuscript No.
(will be inserted by the editor)

Multi-Objective Exploration of Architectural Designs by Composition of Model Transformations

Smail Rahmoun · Asma Mehiaoui-Hamitou · Etienne Borde · Laurent Pautet · Elie Soubiran

Received: date / Accepted: date

Abstract Designing software architectures and optimising them based on extra-functional properties (EFPs) require to identify appropriate design decisions and to apply them on valid architectural elements. Software designers have to check whether the resulting architecture fulfils the requirements and how it positively improves (possibly conflicting) EFPs. In practice, they apply well-known solutions such as design patterns manually. This is time-consuming, error-prone and possibly sub-optimal. Well-established approaches automate the search of the design space for an optimal solution. They are based Model Driven Engineering techniques that formalized design decisions as model transformations and architectural elements as components. Using multi-objective optimizations techniques, they explore the design space by randomly selecting a set of components and applying to them variation operators that include a fixed set of predefined design decisions.

In this work, we claim that the design space exploration requires to reason on both architectural components as well as model transformations. More specifically, we focus on possible instantiations of model transformations materialized as the application of model transformation alternatives on a set of architectural components. This approach was prototyped in RAMSES, a model transformation and code generation framework. Experimental results

Asma Mehiaoui-Hamitou, Etienne Borde, Laurent Pautet
Institut Mines-Telecom;TELECOM ParisTech; LTCI
46 rue Barrault F-75634 Paris Cedex 13
E-mail: firstname.lastname@telecom-paristech.fr

Smail Rahmoun
Institute for Technological Research SystemX
8 Avenue de la Vauve, 91120 Palaiseau
E-mail: firstname.lastname@irt-systemx.fr

Elie Soubiran
Alstom
48 rue Albert Dhalenne 93482 Saint-Ouen cedex
E-mail: firstname.lastname@transport.alstom.com

show the capability of our approach (i) to combine evolutionary algorithms and model transformation techniques to explore efficiently a set of architectural alternatives with conflicting EFPs, (ii) to instantiate, and select transformation instances that generate architectures satisfying stringent structural constraints, (iii) to explore design spaces by chaining more than one transformation. In particular, we evaluated our approach on EFPs, architectures, and design alternatives inspired from the railway industry by chaining model transformations dedicated to implement safety design patterns and software components allocation on a multi-processor hardware platform.

Keywords Component-based software engineering · Model transformations composition · Design space exploration · Rule-based transformation languages · AADL models · Extra-Functional Properties · Multiple Objectives Evolutionary Algorithms · NSGA-II · SAT solvers · Linear Programming

1 Introduction

In the domain of real-time embedded systems, models are used for early validation of extra functional properties (EFPs) such as its timing performance, reliability, power consumption, etc. These models are often built as component-based architectures that provide views on the system. In this context, model transformations play a very important role: a model transformation is a software artefact that specifies a set of actions to generate a target model from a source model. Being executable, these model transformations become reusable solutions to well-defined problems, just like design patterns. Moreover, model transformation alternatives may bring different valid solutions to a given problem and become degrees of freedom in the software architecture design. Since EFPs often conflict with each other (i.e. improving one EFP requires to degrade other EFPs), architectural design usually requires to consider a multi-objective optimization problem with a very large number of potential solutions. As a consequence, producing these solutions with model transformations would require to write many model transformation alternatives, which is impractical. Instead, we have to automate the production of architectural alternatives, and we propose to do so by composition of a small set of transformation alternatives. Indeed, model transformations composition is an interesting technique in our context since it consists in producing new model transformations from a set of existing transformations.

In this paper, we tackle the problem resulting from remarks mentioned above: *how to compose model transformation alternatives to explore a design space made up of architectural solutions with conflicting EFPs?* From our point of view, the application of model transformations to a given set of architecture elements provides the most appropriate expression of degrees of freedom when it comes to design space exploration: (i) the expressivity of model transformation languages help to precisely specify design alternatives, and (ii) the reification of design alternatives with model transformation languages helps to constrain their application in order to reduce the design space by eliminating

incorrect architecture alternatives. This contrasts with existing approaches in which degrees of freedom are extracted from an input architecture solely, and translated into the data structure of an optimization technique: in these approaches, design alternatives are usually under-specified as they depend on the optimization technique and not on a precisely defined model transformation. However, design space exploration based on model transformation alternatives raises difficult issues. Firstly, it is necessary to identify the model transformation alternatives that can be applied to a given set of architecture elements. Then, we have to select which alternative should be applied to each set of architecture elements, while making sure the resulting composite transformations produce correct architectures. A second problem consists to evaluate the impact of composite model transformations on EFPs. This impact is often difficult to estimate before executing the transformation and analysing the resulting architecture. Thirdly, it is important to note that the design space grows rapidly with the number of transformation alternatives and architecture elements. Fourthly, for reuse and maintenance reasons, model transformations are often structured as chains. However, optimising EFPs over a chain of model transformations induces difficult issues. Indeed, the optimisation problem applied to a given link depends on a variable set of architecture elements resulting from the optimisation of the previous link.

To tackle these problems, we propose an approach that automates the composition of model transformation alternatives using Evolutionary Algorithms (EA): such optimisation algorithms enable to explore efficiently large design spaces. This approach was prototyped in RAMSES, and experimental results provided in this paper show the capability of this approach (i) to find near-optimal solutions in a reasonable amount of time, (ii) to limit the combinatorial explosion by excluding a priori transformation compositions producing incorrect architectures, (iii) to explore design spaces by chaining more than one transformation. In addition, architectures evaluation is performed on models produced by these transformations, separating clearly architecture evaluation from its optimisation. This paves the way for reusing architecture evaluation techniques developed separately from the architecture exploration engine.

The remaining of this paper is organised as follows: section 2 gives a precise definition of the problems we address in this paper. In section 3, we present a motivating example that illustrates these problems. An overview of the approach we propose to tackle these problems is given in section 4. Sections 5 and 6 explain our technical contributions in order to combine efficiently EAs and model transformations. In section 7, we present how these contributions were prototyped in RAMSES, the component framework we also used to validate our approach. The corresponding results are provided in section 8. Finally, related works are discussed in section 9 and we give our conclusions in section 10.

2 Objectives and Problem Statement

Several research works [2] propose automated approaches guided by model transformations to explore the design space for optimized solutions. Among these approaches, some of them address the issue of multi-objectives optimization of component architectures with conflicting extra functional properties (EFPs) [1,16]. In these approaches, components selection, components configuration or model transformations are considered as heuristic operators used to explore the design space. From our point of view, such approaches suffer from several drawbacks. We claim that the application of model transformations to a given set of components provides the most appropriate degree of freedom when it comes to design space exploration. To the best of our knowledge, architecture optimization problems have always been studied independently of model transformation techniques. As opposed to existing works, we believe that studying architecture optimizations together with model transformations brings valuable advantages as well as new challenges. In the next subsection, we describe the objectives of our approach with regards to the limitations of existing approaches.

2.1 Optimisation approach on model transformations : objectives

Improving the exploration of the design space. In an approach such as PerOpteryx [16], model transformations are used as heuristic operators to explore the design space. For instance, PerOpteryx produces new architectural solutions by applying a set of model transformations to the previous architectural solutions, each iteration trying to improve EFPs. However, the previous solutions may not match the model transformation preconditions or may produce incorrect solutions. The validation of preconditions is checked dynamically, and the correctness of the produced architecture is made a posteriori, *i.e.* after the application of a transformation. Hence, the exploration becomes even more time-consuming. Firstly, it takes time to evaluate and reject solutions that are a priori invalid. Secondly, the design space is enlarged by a significant number of invalid solutions.

Designing a generic exploration engine based on model transformations. In existing approaches, model transformations are embedded in the exploration engine as heuristic operators for instance. Any new or adapted model transformation designed in a model transformation language has to be implemented in the exploration engine as a specific operator. This translation can be tedious and error-prone. In our approach, we aim at reusing the architecture exploration engine for different model transformations. An architecture exploration engine based on model transformations should take as inputs a set of model transformation alternatives, and a source model. From these inputs, the exploration engine must produce model transformations that generate the best architectures, without modification of the engine source code.

Separation between exploration engine and evaluation tools. In existing approaches, architecture evaluation methods are also embedded in the exploration engine. In our approach, we aim at reusing or separately developing the evaluation methods and the exploration engine. This results from a clear separation between (i) the architecture exploration engine, based on model transformation alternatives, and (ii) the architecture evaluation, based on models produced by these transformations. Of course, evaluation techniques and transformations are coupled: the evaluation is based on analysis of models produced by the transformations. However, they can be developed separately and existing evaluation tools, when existing, can be reused.

Optimisation of chained model transformations. Optimizing a chain of model transformations is a difficult problem. It cannot be reduced to chaining the optimisation method on individual model transformations. Firstly, an optimization step produces an a priori unknown number of architectural components for the next optimization step. Secondly, the impact of a design decision taken in each step can only be analysed on solutions that contain all these decisions (*i.e.* evaluation of EFPs on intermediate models is not possible). In our approach, we aim at optimising a chain of model transformations by iterating the exploration methods over the different links of the chain and evaluating only the final architectural model.

To the best of our knowledge, these objectives are not fulfilled in existing architecture optimization methods. The main reason is that architecture optimizations are usually studied in isolation of model transformations. In next subsection, we present the challenges that have to be faced when designing architecture optimization techniques based on model transformations.

2.2 Architecture optimisation based on model transformations: problems

When combining optimization and model transformations techniques, the first problem is to identify the variables (also called degrees of freedom) of the optimization problem from a set of model transformation alternatives:

Problem 1. How to extract a representation of model transformation alternatives, and map this representation into an optimization method?

In addition, architecture optimizations are known to require the exploration of an important number of design alternatives. To reduce this combinatorial explosion problem, we must ensure that the optimization framework only explores correct architectures. In this work, an architecture is considered to be correct if it respects a set of predefined structural constraints (potentially expressed using dedicated languages such as OCL [21]). Since architectures are produced by executing composite model transformations, we have to answer to the following problem:

Problem 2. How to ensure, possibly before executing it, that a composite model transformation will produce an architecture respecting a set of predefined structural constraints?

In an architecture optimisation problem, the objective is to produce and select the best architectures with respect to a set of EFPs. When candidate architectures are produced by composite transformations, solving the optimisation problem boils to select the transformation having the best impact on EFPs. However, this impact is often difficult to compute a priori, and it is even more difficult to compute when dealing with composite model transformations produced automatically.

Problem 3. How to evaluate composite transformations with respect to their impact of EFPs?

In order to ease their maintenance and improve their reusability, model transformations are often structured as model transformation chains [10]. However, most of the time, the intermediate architecture produced by a link of the chain does not contain all the necessary data to evaluate EFPs. In this case, architectures exploration cannot be done with isolated model transformations, but must consider chains of model transformation alternatives.

Problem 4. How to explore a bi-dimensional design space, made up of transformation chains with transformation alternatives for each link of the chain?

In the next section, we present a motivating example which illustrates the problems identified above.

3 Motivating example

The motivating example we present in this section is based on the definition of AADL[1] models [12], AADL-to-AADL model transformations, and EFPs analysis techniques.

3.1 AADL models

AADL is a standardized architecture description language that aims at supporting the design, analysis and integration of real-time embedded systems [12]. An AADL system is described as an assembly of software components (*e.g.* process, thread, data, subprogram) and hardware components (*e.g.* processor, memory, bus, device). Interfaces of components are described with predefined port types (*e.g.* data, event, or event data ports), or component accesses (*e.g.* data or subprogram accesses for software components, bus or memory accesses for hardware components). The internal structure of a component is described by a set of connected subcomponents: connections among interfaces of subcomponents (or between interfaces of a subcomponent and its enclosing component) define communication channels between components. Standardized or user-defined properties decorate AADL models to precise the characteristics of the component-based architecture. The next paragraphs detail the constituents of the hardware and software models we consider.

[1] Architecture Analysis and Design Language

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 1 (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) which connects these processors.

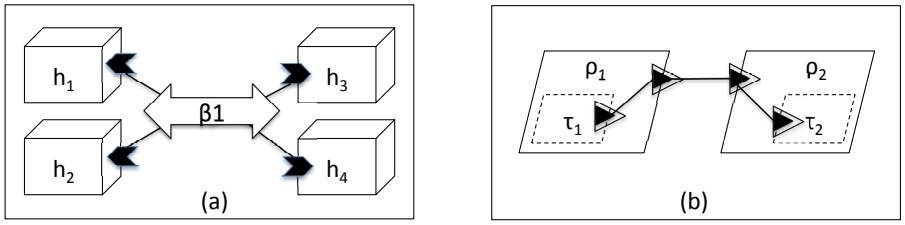

Fig. 1: (a) hardware and (b) software AADL models

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}. $Proc(\tau_i)$ 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}. Threads and messages are, respectively, characterized by a vector of worst-case execution times (WCETs) $\boldsymbol{\omega_{\tau_i}} = \{\omega_{\tau_i,h_1}, \omega_{\tau_i,h_2}, ..., \omega_{\tau_i,h_{Nc}}\}$ and worst-case transmission times (WCTTs) $\boldsymbol{\omega_{\mu_i}} = \{\omega_{\mu_i,\beta_1}, \omega_{\mu_i,\beta_2}, ..., \omega_{\mu_i,\beta_{N\beta}}\}$, 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 1 *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 1 (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 $\Gamma_1 = [\tau_1, \mu_1, \tau_2]$.

In the next subsection, we describe the model transformation alternatives we consider to improve EFPs of such architectures.

3.2 AADL-to-AADL model transformations

Model transformations described in this subsection implement classical design decision in critical embedded systems: the replication of software components, and their allocation onto hardware components.

3.2.1 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)*.

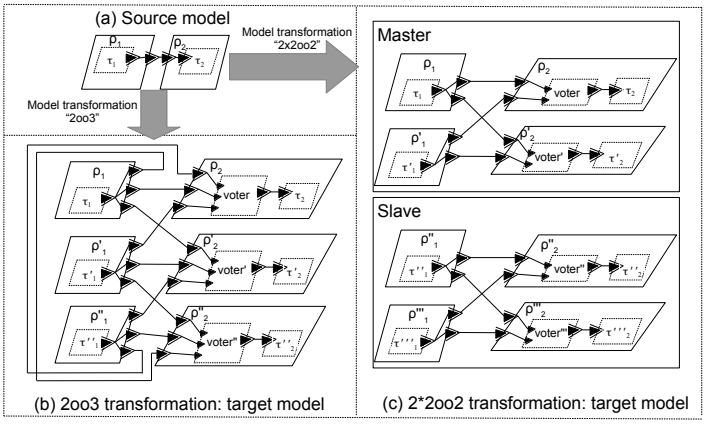

Fig. 2: Application of replication model transformations

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 operating in parallel. Communications between processes are also replicated such that each process and each of its replicas communicates with successor processes and all of their replicas. Figure 2 (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 2 (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.

3.2.2 Software components allocation

Software **components allocation** 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.

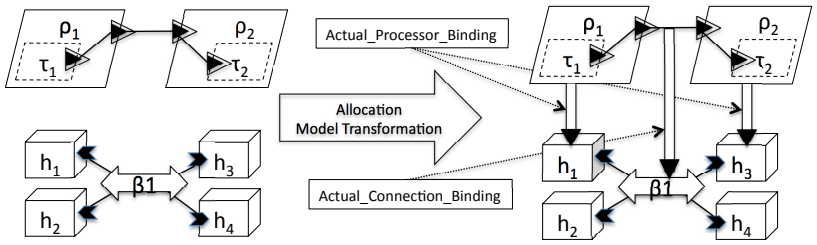

Fig. 3: Example of allocations of software components

Figure 3 illustrates a very 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, the 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). Once the allocation of software components on hardware components is performed, several EFPs can be evaluated as explained in next subsection.

3.3 EFP analysis

In this subsection, we describe the analysis methods we use in this paper to evaluate two important EFPs of critical embedded systems: end-to-end response time and reliability.

3.3.1 End-to-end response time analysis

For a given path (see definition 1), end-to-end response time represents the highest amount of time required for an information to transit from the source of the path, until the destination of the path. 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 [27].

Response time of threads. Worst case response time R_{τ_i,c_j} of a periodic thread τ_i, executed on processor h_j, is represented with R_{τ_i,h_j} and computed according to (1). B_{τ_i} is the blocking time of the thread τ_i, which depends on shared resources accessed by the thread and the way in which the shared resources are protected from multiple accesses. In our study, messages transmitted among tasks on the same processor represent shared resources are protected with the Priority Ceiling Protocol (PCP) [23]. To compute the blocking time with the PCP we refer to equation (2), where $\omega'_{\tau_i,\mu_k,c_j}$ is the WCET of a thread τ_i for accessing (reading/writing) a shared message μ_k, executing critical section on c_j. The priority ceiling of this message is $PC(\mu_k)$.

$$R_{\tau_i,c_j} = \omega_{\tau_i,c_j} + \sum_{\tau_k \ s.t. \ (\pi_{\tau_k} > \pi_{\tau_i})} \left\lceil \frac{R_{\tau_i}}{\varphi_{\tau_k}} \right\rceil * \omega_{\tau_k,c_j} + B_{\tau_i,c_j} \quad (1)$$

$$B_{\tau_i,c_j} = \max_{\mu_k \in M, \tau_j \in T} \omega'_{\tau_i,\mu_k,c_j} \ s.t. \ \pi_{\tau_i} \leq PC(\mu_k) \ and \ \pi_{\tau_i} > \pi_{\tau_j} \quad (2)$$

Response time of messages. Worst case response time R_{μ_i,β_j} of message μ_i transmitted on bus β_j is computed according to an equation similar to equation (1) except that scheduling of messages is non-preemptive.

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 of all threads and messages, as well as the periods of all the messages and their receiver thread on the path (see equation (3))[28].

$$L_{\Gamma_i} = \sum_{\tau_j \in \Gamma_i} R_{\tau_j} + \sum_{\mu_j \in \Gamma_i} R_{\mu_j} + \varphi_{\mu_j} + \varphi_{Dest(\mu_j)} \quad (3)$$

3.3.2 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 1) in 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:

$$R(\Gamma_i) = \prod_{h_i \in PS(\Gamma_j)} R(h_i) \tag{4}$$

Given the replication transformations introduced in previous subsection, the target models of these transformations will 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:

$$\begin{aligned} R^{2oo3}(\tau_s, \tau_d) = & \prod_{\Gamma_i \in SE^{2oo3}} R(\Gamma_i) \\ & + R(\Gamma'_{s,d}) * R(\Gamma''_{s,d}) * (1 - R(\Gamma'''_{s,d})) \\ & + R(\Gamma''_{s,d} * R(\Gamma'''_{s,d}) * (1 - R(\Gamma'_{s,d})) \\ & + R(\Gamma'_{s,d}) * R(\Gamma'''_{s,d}) * (1 - R(\Gamma''_{s,d})) \end{aligned} \tag{5}$$

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).

In next subsection, we discuss the adequacy between (i) the type of architectures, EFPs, and design alternatives we consider in this paper and (ii) the problems described in section 2.

3.4 Compliance between the example and the problems

Firstly, this example illustrates that design patterns selection cannot be done in isolation for model elements: for achieving a correct voting mechanism, applying the 2oo3 pattern to a process ρ requires to apply the same replication pattern to all the processes ρ is connected to. This was implicitly the case on figures 2 (b) and 2 (c), where the replication pattern applied to ρ_1 be composed with the same replication pattern applied to ρ_2. In addition, replicated processes should be allocated to different processors (otherwise the replication pattern is useless). This example thus illustrates problem 1: a composite transformation must produce architectures that enforce the respect of predefined architectural constraints.

Secondly, the chosen design pattern impacts differently EFPs of interest: one is better for reliability but incurs more communications, and vice-versa. Of course, software components allocation also impacts EFPs of interest, which illustrates problem 2: it is difficult to anticipate, a priori, the impact of a transformation made up of both the selection of a design pattern and the allocation of software components on hardware components.

Thirdly, this example enables to illustrate the combinatorial complexity of design space exploration (cf. problem 3): for a simple model with six process components and four processors, there are $2^6 = 64$ choices for safety design pattern and, depending on the processes safety design pattern, there are at least $(6 * 3)^4 = 104976$ and at most $(6 * 4)^4 = 331776$ possible allocations.

Fourthly, one easily understands from the description of the safety design patterns above, that the choice of a design pattern must be performed **before** the components allocation. Besides, both reliability and end-to-end data flow analysis can only be performed once software components have been allocated to hardware components (illustration of problem 4, explained in section 2).

In the next section, we explain our approach to tackle these problems.

4 Proposed Approach

This section gives an overview of our approach for design space exploration based on model transformation alternatives. In subsection 4.1, input artefacts of our approach are described. Then, subsection 4.2 gives an overview on the process we propose to address research problems identified in section 2.

4.1 Input artefacts

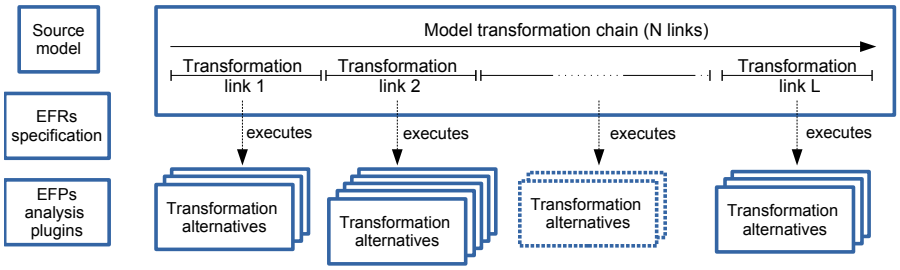

Fig. 4: Input artefacts of the approach

Figure 4 lists the input artefacts of our approach:

1. *The source model*, produced by the system designers, models the architecture of the system under design.
2. *Extra functional requirements (EFRs) specification* defines the acceptable limits for EFPs of the system, as well as objectives in terms of maximization or minimization of these EFPs. We expect these EFPs to be conflicting, as this is the case in most design space exploration problems. This artefact is filled by EFPs experts, or by the system designers.

3. *EFPs analysis plugins* are software artefacts capable evaluating EFPs of a given architecture. These plugins are reused, adapted, or created by the model transformation framework provider.
4. *The model transformations chain*, made up of a predefined ordered set of L transformation links. In this chain, each link references a limited set of model transformation alternatives. This artefact is provided with the model transformation framework.

Given these different artefacts, our objective is to find a model transformation chain that answers at best the trade-off among conflicting EFPs of the system. To reach this objective, we propose a design space exploration process, as described in next subsection.

4.2 Design space exploration process

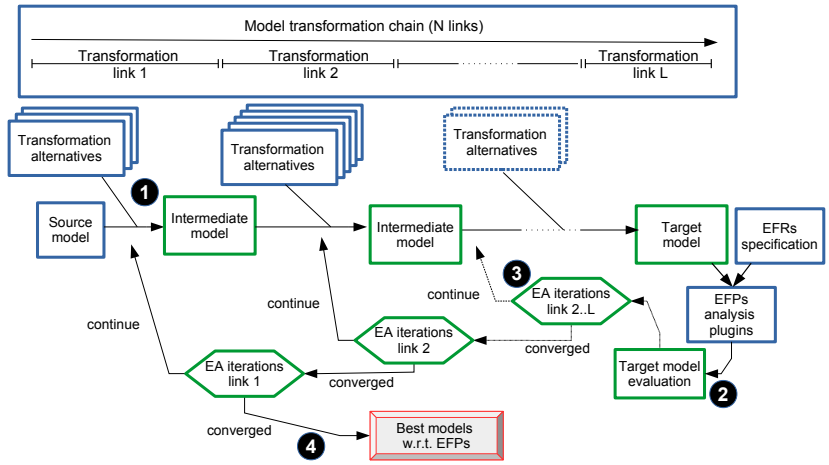

Fig. 5: Approach Overview

Figure 5 gives an overview of the process we propose in this paper: from the definition of a model transformation 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, and repeated for each link of the transformation chain until the target model is produced. The composition mechanism, used in this step of the process, is explained in section 5. It answers problems 1 and 2, identified in section 2.

We assume, in this approach, that intermediate models do not contain all the information required to analyze EFPs of the architecture: intermediate

models are enriched in each link of the transformation chain. Once produced, the target model is analyzed with respect to EFPs (bullet 2 in figure 5), and the analysis results are used to evaluate composite transformations. This part of the process answers problem 3, identified in section 2.

The process we propose is iterative: each iteration produces, executes, and evaluates a (sub)chain of composite transformations. In addition, because of the combinatorial complexity 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 in figure 5). The data structure used to compose model transformation with EAs is explained in section 6. In addition, we can see in figure 5 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. This part of the process answers problem 4, identified in section 2.

When the top-level loop has converged, the exploration process stops and returns the best models that were found (see bullet 4 in figure 5).

4.3 Model Transformation Composition with EAs: challenges

The most challenging steps, in this process, are those identified with bullets 1 and 3 in figure 5: in order to explore the design space efficiently, we need to produce only correct architectures (see definition 2 below). As a consequence, our composition mechanism must produce only valid transformation (see definition 3 below).

Definition 2 *A **correct architecture** is a component-based architecture that respects a set of structural constraints and conforms to requirements defined by software designers.*

Definition 3 *A **valid transformation** is a transformation that, once executed, produces a correct architecture (see definition 2).*

As far as structural constraints are concerned, we aim at validating them **a priori**, i.e. before executing the transformation. As far as EFPs are concerned, we aim at validating them **a posteriori**, i.e. after executing the transformation. To reach the objective of **a priori** validation, we need to address the following two challenges:

1. From a set of alternative transformations, how to produce composite transformations that are **a priori** valid? To address this challenge, we propose to combine rule-based model transformation techniques and boolean satisfiability techniques. This contribution is explained in section 5.

2. How to make sure the application of EAs also produces composite transformations that are **a priori** valid? To address this challenge, we structure genomes of a population in order to make sure the application of genetic operators always produce new composite transformations that are a priori valid. This contribution is explained in section 6.

To the best of our knowledge, this work presents the first model-based architecture optimization framework, enforcing **a priori** that explored architecture respect predefined structural constraints. It is also the first attempt to combine model transformation techniques, including model transformation chains, and multi-objective optimization techniques.

In the next section, we present our technical solution to produce **a priori** valid composite transformations from a set of transformation alternatives.

5 Production of A Priori Valid Composite Transformations

In order to produce composite transformations from a set of transformation alternatives, we propose an approach based on rule-based model transformations, and SAT solving techniques. In this approach, SAT is used in order to ensure we only produce composite transformations that are **a priori** valid. Note that this approach is generic: it is defined independently of a specific architecture optimization problem. We explain this contribution in the remainder of this section, starting from the definition of rule-based model transformations:

Definition 4 *A **rule-based model transformation** is a model transformation made up 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).*

Given this definition, rule-based model transformations instantiation produces, 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;*
3. *A_i is the set of actions that R executes when it is applied to E_i.*

In order to produce composite transformation, we should select a valid set of TRIs. Confluence of these TRIs is a first property to consider, as explained in next subsection.

5.1 Confluent Transformation Rules Instantiations

TRIs cannot be selected randomly to produce composite transformations. Indeed, the execution semantics of rule-based model transformation languages [13], requires transformation rules to be executable in any order without modifying the result of the transformation. This characteristic is called confluence. We thus need to identify confluent transformation rule instantiations (see definition 6).

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.*

To build a composite transformation by selection of TRIs, we should select confluent TRIs only. In the remainder of this paper, we use the notation $TRI_i \oplus TRI_j$ to express that rule instantiations TRI_i and TRJ_j are **not** confluent. In other words, they are exclusive. In our approach, we propose to encode the detection of non-confluence with two types of situation:

1. non-confluence exists when more than one rule can be applied to the same tuple of elements in the source model. Formally, this means :

$$\begin{aligned} &\exists (R, R')\ s.t.\ R \neq R' and\ TRI_i =< R, E_i, A_i > \\ &\quad and\ TRI_j =< R', E_j, A_j >\ and\ E_i = E_j \end{aligned} \tag{6}$$

2. non-confluence 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:

$$\begin{aligned} &\exists (E_i, E_j)\ s.t. E_i \neq E_j\ and\ TRI_i =< R, E_i, A_i > \\ &and\ TRI_j =< R, E_j, A_j >\ and\ US(TRI_i) = US(TRI_j) \end{aligned} \tag{7}$$

When producing a composite transformation by selecting TRIs, the resulting set of TRIs must be confluent. Given the definition of confluent transformation rule instantiations, and the definition of validity constraints on the selection of TRIs, we formalize the selection of TRIs as follows.

5.2 Formalization of TRIs selection

We define S, a **Set of P Non-confluent Transformation Rule Instantiations** $S = \{TRI_i\}_{i=1..P}$ such that:

$$\forall i, j \in [1, P], i \neq j, TRI_i \in S\ and\ TRI_j \in S\ \Rightarrow TRI_i \oplus TRI_j \tag{8}$$

Considering this definition, for each set of non-confluent transformation rule instantiations $S = \{TRI_i\}_{i=1..P}$, the selection of TRIs may be decomposed into the following formulas:
1) AtLeastOne, dedicated to ensure that at least one of the non-confluent TRIs is selected:

$$AtLeastOne(S) = \bigvee_{i=1}^{P} Sel(TRI_i) \tag{9}$$

2) AtMostOne, dedicated to ensure that at most one of the non-confluent TRIs is selected:

$$AtMostOne(S) = \bigwedge_{i=1,j=1,i\neq j}^{i=P,j=P} \neg(Sel(TRI_i) \wedge Sel(TRI_j)) \tag{10}$$

Combining equations (9) and (10), we obtain *SelectOne*, dedicated to select exactly one TRI from S:

$$SelectOne(S) = AtLeastOne(S) \wedge AtMostOne(S) \tag{11}$$

The selection of a set of TRIs satisfying equation (11) is guaranteed to be confluent. In addition to the confluence property, there might exist dependencies among TRIs: selecting a TRI may require to exclude or select another TRI. Besides, we should ensure that the composite transformation, resulting from the selection of TRIs, produces an architecture that respect structural constraints. We enforce the respect of these constraints by (i) formalizing them as validity constraints on the selection of TRIs, and (ii) using SAT solving techniques to find confluent TRIs enforcing these constraints. Thus, we translate structural constraints expressed on component-based architectures into validity constraints on the composition of transformation alternatives.

5.3 Formalization of Validity Constraints on TRIs Selection

We propose to formalize constraints on the selection of TRIs with a set of boolean formulas. We consider $T = \{TRI_i\}_{i=1..N}$ a set of transformation rule instantiations, and the function Sel defined hereafter:

$$\begin{array}{lrl} Sel: & T & \rightarrow \mathbb{B} = \{True, False\} \\ & TRI & \rightarrow b, \text{ where } b \text{ is } True \text{ if } TRI \text{ should be included, and } False \\ & & \quad \text{if } TRI \text{ should be excluded from } T. \end{array}$$

Based on these definitions, we propose to express validity constraints on the selection of TRIs as a conjunction of constraint on the Sel function:

$$ValidityConstraints = \bigwedge_{i=1}^{N} (Sel(TRI_i) \Rightarrow BoolExpr(T_i)) \tag{12}$$

where T_i is a subset of $T - \{TRI_i\}$, and $BoolExpr$ is a boolean expression over $TRIs$ in T_i, using (i) the Sel function, (ii) simple boolean operators *and*, *or*, and *not* ($\wedge$, $\vee$, and $\neg$).

For instance, consider T a set of three TRIs, validity constraints may be expressed as a conjunction of the following boolean expressions:
$Sel(TRI_1) \Rightarrow (Sel(TRI_2) \wedge \neg Sel(TRI_3))$
$\vee(\neg Sel(TRI_2) \wedge Sel(TRI_3))$
$Sel(TRI_2) \Rightarrow true$
$Sel(TRI_3) \Rightarrow true.$
In this example, selecting TRI_1 implies to select either TRI_2 or TRI_3 (but not both). Selecting TRI_2 or TRI_3 has no further implications.

Finally, for all the sets of non-confluent rule instantiations $S_{ii\in[0..Q]}$, the selection of a valid set of TRIs boils to evaluate the satisfiability of the boolean formula:

$$\bigwedge_{i=1}^{Q}(SelectOne(S_i)) \wedge ValidityConstraints \tag{13}$$

A valid composite transformation is thus defined by a a function Sel that satisfies equation (13). SAT techniques can thus be used to produce composite transformations by selection of a set of TRIs. In addition, such composite transformations are **a priori** valid since structural constraints on the component-based architecture have been translated into validity constraints on the selection of TRIs. However, the number of potential solutions grows rapidly with the number of non-confluent TRIs: considering M sets of non-confluent TRIs (as defined in equation (8)), each of them having N TRIs, the number of composite model transformations is at most M^N. For this reason, in most cases, enumerating and evaluating each composite transformation is not feasible. In order to automate the exploration of the design space generated by model transformation alternatives, we propose to rely on Evolutionary Algorithms (EAs). In the next section, we explain how we propose to structure composite transformation in order to use EAs.

6 Evolutionary Algorithms with Chains of Model Transformations

Inspired from genetics, EAs are based on the following principles: from an initial population of genomes, select the best genomes regarding a set of objective functions (*e.g.* EFPs in our case), and produce a new population using a selection operation and genetic operators (*i.e.* mutation, crossover). In EAs, each genome is made up of a fixed size ordered set of genes. Traditionally, these genomes are represented by a binary encoding (ordered sequence of fixed size, of 0 and 1), but other encodings are also possible. In our approach, we propose an encoding that aims at producing new composite transformations by applying genetic operators on composite transformations. More importantly, composite transformations resulting from the application of genetic operators must be **a priori** valid. To the best of our knowledge, this is the first work ensuring the a priori validation of structural constraint while applying EAs for architecture optimization. Beside, it provides a generic exploration engine: the algorithm operates on data structures defined independently of a specific

architecture optimization problem. In this section, we explain our encoding of composite model transformations as genomes, and their usage in EAs.

6.1 Genomes encoding

Traditionally, the initial population of an EA is created by elaborating *randomly* a set of genomes. A simple way to do this is to assign a random value of each gene of each genome. In our work, a genome is a composite transformation, made up of TRIs. However, these TRIs cannot be selected randomly as explained in section 5: selected TRIs must satisfy constraints defined in equation (13). Selecting *randomly* TRIs that satisfy such constraints may be time consuming, and may even not converge. In case it converges, another issue is the difficulty to apply genetic operators while preserving the satisfaction of these constraints. To overcome these issues, we propose an encoding of genomes into **a priori** valid transformation for both the initialization step and the application of genetic operators. We explain this encoding in the next paragraphs, and the application of genetic operator in following subsections.

Our solution is to group $TRIs$ involved in the same validity constraints into **partitions**, and to extract, from each partition, a **pool** of interchangeable atomic transformation instantiations (see definition 7). Then, we create a composite transformation by selecting randomly one atomic transformation instantiation from each **pool**.

Definition 7 *An **atomic transformation instantiation** is a transformation made up of a set of transformation rule instantiations that cannot be decomposed into smaller transformations.*

Our objective is to group $TRIs$ involved in the same validity constraints into **partitions**. To do so, we first reorganize equation (13) under a conjunctive normal form. We call B the set of boolean expressions 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, and we note β_q the boolean formula corresponding to the q^th part of the partition.

Let consider the following boolean expression: $[Sel(TRI_3)] \wedge [Sel(TRI_1) \vee Sel(TRI_2)] \wedge [Sel(TRI_1) \vee Sel(TRI_4)]$. Its partition leads to parts: $\{Sel(TRI_3)\}$ and $\{(Sel(TRI_1) \vee Sel(TRI_2)), (Sel(TRI_1) \vee Sel(TRI_4))\}$ represented by boolean expressions: $\beta_1 = Sel(TRI_3)$ and $\beta_2 = (Sel(TRI_1) \vee Sel(TRI_2)) \wedge (Sel(TRI_1) \vee Sel(TRI_4))$.

A pool of atomic transformation instantiations is finally produced automatically by instantiating, for each equation β_q, all the combinations of boolean values for $Sel(TRI_i)$ that satisfy β_q. The solution we propose to achieve that is to use the SAT solving techniques. Applying this technique for each equation β_q, we produce a set of solutions satisfying the boolean formula defined in equation 13.

For instance, applying SAT to β_1 leads to one atomic transformation instantiation (ATI): $ATI11 = \{TRI_3\}$, and applying SAT to β_2 leads to a pool of several atomic transformation instantiations: $ATI21 = \{TRI_1\}$, $ATI22 = \{TRI_2, TRI_4\}$, $ATI23 = \{TRI_1, TRI_2, TRI_4\}$. We then structure a composite transformation (CT) by choosing, for each equation β_q, one atomic transformation instantiation in the corresponding (q^{th}) pool: $CT_1 = \{ATI_{11}, ATI_{21}\}$, $CT_2 = \{ATI_{11}, ATI_{22}\}$, etc.

Based on this new formulation of equation (13), we define a genome (or composite transformation) as an ordered set of atomic transformation instantiation: $CT = \{ATI_{ij}\}_{i=1..n, j \in \mathbb{N}}$, where i is an identifier of a pool of interchangeable atomic transformation instantiations and j is the identifier of one element in this pool. Using this representation, we can produce an initial population by selecting randomly an atomic transformation instantiation from each pool. The resulting composite transformation are, **a priori**, valid.

6.2 Genetic operators applications

Using the representation defined in previous subsection (6.1), we can easily encode composite transformations for EAs by (i) representing composite transformations as genomes, where each genome is composed of an ordered sequence of genes: atomic transformation instantiation; (ii) selecting transformation alternatives according to their $EFPs$ evaluation; and (iii) mixing their genes (crossover) and/or creating new genomes (mutation) while producing valid transformations. In the next paragraphs, we present the genetic operators we used on composite transformations.

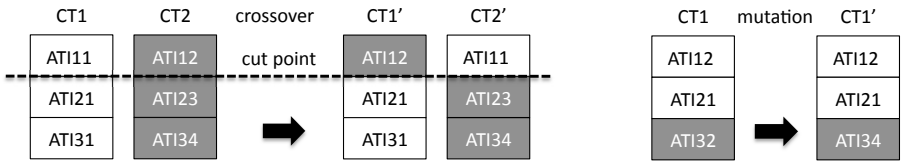

Fig. 6: Crossover and Mutation Operators

6.2.1 Crossover

The crossover consists in producing new genomes (called off-springs) by mixing the genes of two existing genomes (called parents). Two off-spring solutions are 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. Using our representation of composite transformations as an ordered set of atomic transformation instantiations, applying the crossover operator becomes easy: wherever we apply the cut in a

composite transformation, we always have a solution that satisfies the validity constraints. Figure 6 illustrates the application of the crossover operator on two genomes CT_1 and CT_2.

6.2.2 Mutation

After performing the crossover operator, the obtained genomes are mutated. The mutation operator selects a gene (randomly), and changes its values. In figure 6, the mutation operator acts on a single genome CT_1 to obtain a mutated genome CT_1' by replacing an atomic transformation instantiation by another one from the same pool. As a consequence, the resulting composite transformation is **a priori** valid.

6.3 Identifying Pareto-optimal solutions using NSGA-II

To finalize our approach, we combine all the steps defined above, and integrate them in an evolutionary algorithm. In this paper, we propose to use NSGA-II: an improved version of NSGA [24] (Non-dominated Sorting Genetic Algorithm). NSGA-II operates as follow:

1) First, create a random population P_t composed of N (specified by end-users of our method) CT. Execute and evaluate each CT regarding $EFPs$ by evaluating the corresponding target model.

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

3) Create an off-spring population Q_t of size N using selection tournaments, crossovers and mutations. We apply each CT of Q_t to the source model and evaluate the $EFPs$ of the target models.

4) 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) Sort R_t with a fast non-dominated sorting procedure [8], to identify non-dominated fronts $\{F_i\}_{i=1..N}$ (F_1 being the best and F_N the worst front).

6) Generate the new parent population P_{t+1} of size N. The solutions belonging to the best non-dominated front, (i.e. F_1) represent the best solutions in R_t and must be emphasized more than any other solution. 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 solutions of F_2 are chosen next. In some cases, not all the solutions 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 solutions in F_i. In order to choose which ones are selected, these solutions are sorted according to their crowding distance [8], and the solutions 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 apply this process to the next generation until a stopping criterion is met.

We presented an automatic approach to select and compose model transformation alternatives using $NSGA-II$. We present in the next subsection the application of our approach on chains of model transformations.

6.4 Exploration of chained model transformations

When the model transformation we consider is structured as a chain of model transformation alternatives, we propose to apply the principles of EAs to each link of the chain, aiming at producing a non-dominated population of size N given by the end-user of our approach. Considering L links, as illustrated on figure 5, the exploration process should produce $L-1$ sets of intermediate models, and 1 set of N target models. Target models are produced by the L^{th} link by applying the EA on the population made up of composite transformations applied on the $(L-1)^{th}$ intermediate model.

We first produce this $(L-1)^{th}$ intermediate model by selecting randomly a valid composite transformation for each of the previous links (links 1 to $L-1$). We then explore composite transformations for the L^{th} link, using EA iterations, as illustrated in figure 5. Note that each link l has its own convergence criteria for an EA such as the maximum number of models that may be produced without modifying the set of non-dominated solutions or the maximum number of models produced in iteration l.

We thus iterate on the L^{th} link of the transformation chain until a convergence criterion of the EA is reached for this link. Then, the process continues with link $L-1$: from the $(L-2)^{th}$ intermediate model, new models are created (i) first randomly until pop_{L-1} models have been obtained for the next link (the L^{th}) (ii) and then using EAs operators until the convergence criterion has been reached for the $(L-1)^{th}$ link. Note that, for evaluating created populations for a given link, the target model needs to be produced and its EFPs evaluated.

The process continues iteratively until the convergence criterion of link 1 has been reached and a set of non-dominated target models has been obtained. Of course, the amount of evaluated architectures and thus the time needed for convergence highly depends on parameters given by end-users. These parameters are set by end-users of our approach to either increase or decrease the number of composite transformations produced for each link of the transformation chain.

In the next sections, we describe model transformation alternatives we implemented when prototyping our approach in the RAMSES framework [5].

7 RAMSES: Prototyping the Approach with ATL Transformations

The method we propose in this paper has been implemented and experimented in RAMSES, a component-framework for critical embedded systems [5]. RAMSES helps to design software applications for embedded systems by implementing AADL-to-AADL model transformations written in ATL. In this section,

we first present ATL model transformations implementing the replication patterns described in section 3. Then, we explain how these transformations are formalized as alternatives with validity constraints on their composition.

7.1 ATL Model Transformations

ATL is a hybrid transformation language (*i.e.* a transformation language with both rules and imperative statements) which execution relies mainly on a pattern matching semantics [15]: in ATL, a transformation consists of a set of declarative *matched rules*, each specifying a *source pattern* and a *target pattern*. 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 [21] 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 target objects.

Listing 1: ATL rule for 2*2oo2: processes replication

```
1  rule m_Process_2_2oo2
2  {
3    from
4      c : AADLI!ComponentInstance (c.category = #process)
5    to
6      proc1_22oo2: AADLBA!ProcessSubcomponent (...),
7      proc2_22oo2: AADLBA!ProcessSubcomponent (...),
8      proc3_22oo2: AADLBA!ProcessSubcomponent (...),
9      proc4_22oo2: AADLBA!ProcessSubcomponent (...)
10 }
```

Listings 1 and 2 provide snippets of the ATL code used to implement the 2*2oo2 model transformation alternative for components replication. This transformation alternative was already explained in section 3.2, and is depicted in figure 2. In listing 1, ATL rule `m_Process_2_2oo2` transforms every AADL process component into four process components identified with the following *target object* identifiers: `proc1_22oo2`, `proc2_22oo2`, `proc3_22oo2`, and `proc4_22oo2`. For the sake of concision, this listing does not develop the creation of these processes. Given the execution semantics of ATL, this rule will match any AADL component instance of the process category, thus producing for each process ρ_i of our software model, a trasfromation rule instantiation $TRI_i =< m\_Process\_2\_2oo2, \{\rho_i\}, \{create(procSub1), ...\} >$.

In listing 2, ATL rule `m_PortConnection_2_2oo2` transforms connections among process components in the source model, into connections among their replicas in the target model. As one can see in figure 2, a connection between process components in the source is transformed into 8 connections in the target model. This is represented in rule `m_PortConnection_2_2oo2` with the creation of `cnx1_1_22oo2`, `cnx1_2_22oo2`, etc. For the sake of concision, the creation of only one of the eight connections is fully developped in this listing. This rule will match, in the source model of the transformation, any connection cnx_c between two process components, thus producing a transformation rule instantiation:

$TRI_c =< m\_PortConnection\_2\_2oo2, \{cnx_c\}, \{create(cnx1\_1\_22oo2), ...\} >$.
In `m_PortConnection_2_2oo2`, one may note the usage of `resolveTemp`, an ATL mechanism that enables to retreive elements created in `m_Process_2_2oo2`. The use of this mechanism induces a dependency between the application of rules `m_Process_2_2oo2` and `m_PortConnection_2_2oo2`. This dependency is captured in the transformation rules catalog, as explained in the next section.

Listing 2: ATL rule for 2*2oo2: connections replication

```
1  rule m_PortConnection_2_2oo2
2  {
3   from
4     cnx: AADLI!ConnectionReference(cnx.isProcessPortsConnection())
5   using
6   {
7     cSrc : AADLI!ComponentInstance = cnx.getSrcCptInstance();
8     cDst : AADLI!ComponentInstance = cnx.getDstCptInstance();
9   }
10  to
11    -- feature f_1: PROC_1_src -> PROC_1_dst --
12    cnx1_1_22oo2: AADLBA!PortConnection (
13      name <- cnx.getName()+'1_1',
14      source <- sourceCE1_1,
15      destination <- destinationCE1_1
16    ),
17    sourceCE1_1: AADLBA!ConnectedElement (
18      connectionEnd <- cnx.source,
19      context <- thisModule.resolveTemp(cSrc, 'proc1_22oo2')
20    ),
21    destinationCE1_1: AADLBA!ConnectedElement (
22      connectionEnd <- cnx.destination,
23      context <- thisModule.resolveTemp(cDst, 'proc1_22oo2')
24    ),
25    -- feature f_1: PROC_1_src -> PROC_2_dst --
26    cnx1_2_22oo2: AADLBA!PortConnection
27    ...
28    -- 6 other connections ommitted for the sake of concision
29  }
```

7.2 Transformation Alternatives Specification

RAMSES developers rely on an internal language in order to specify model transformation alternatives and validity constraints on their compositions in a Transformation Rules Catalog (TRC). Listing 3 provides a subset of the TRC we used to describe transformation alternatives for components replication: *2oo3* and *2*2oo2*. This TRC is made up of two main parts:

1. a description of model transformation alternatives, from line 1 to line 10, lists the set of ATL modules and rules being part of each alternative. The *2*2oo2* alternative is made up of transformation rules described in the previous section. The *2oo3* alternative is made up of very similar transformation rules: `m_Process_2oo3` and `m_PortConnection_2oo3`. Rule `m_Process_2oo3` (respectively `m_PortConnection_2oo3`) has the very same input pattern as `m_Process_2_2oo2` (*resp.* `m_PortConnection_2_2oo2`) but generates three replicas (*resp.* 9 connections among replicas).

2. a specification of validity constraints, from line 12 to line 30. The first one, from line 14 to line 21, specifies that when `m_PortConnection_2_2oo2` is applied on a connection (identified as `cnx` in the constraint), it is necessary to apply rule `m_Process_2_2oo2` on both ends of the connection (retreived executing a OCL helpers called `getDestinationProcess` and `getSourceProcess` on `cnx`). The second constraint, from line 22 to line 27, specifies that when applying `m_Process_2_2oo2` on a process component `processInstance`, `m_PortConnection_2_2oo2` should be applied on all the connections having `processInstance` as a source (retreived by applying the OCL helper `getSourceConnectionReference` on `processInstance`). Gathering these two constraints lead to ensure that the 2*2oo2 alternative is applied to sets of interconnected process components. Very similar constraints are expressed for the application of the 2oo3 alternative in the remaining of the TRC.

```
1  Modules
2  {
3    2_2oo2.atl: m_Process_2_2oo2, m_PortConnection_2_2oo2;
4    2oo3.atl: m_Process_2oo3, m_PortConnection_2oo3;
5  }
6
7  Alternatives {
8    replication_2_2oo2 { modules: 2_2oo2.atl },
9    replication_2oo3 { modules: 2oo3.atl }
10 }
11
12 Constraints {
13   // 2*2oo2
14   Apply(replication_2_2oo2.m_PortConnection_2_2oo2, {cnx})
15   [
16     requires (replication_2_2oo2.m_Process_2_2oo2,
17       {getSourceProcess(cnx)}
18     ) and requires (replication_2_2oo2.m_Process_2_2oo2,
19       {getDestinationProcess(cnx)}
20     )
21   ];
22   Apply(replication_2_2oo2.m_Process_2_2oo2, {processInstance})
23   [
24     requires (replication_2_2oo2.m_PortConnection_2_2oo2,
25       {getSourceConnectionReference(processInstance)}
26     )
27   ];
28   // similar constraints for 2oo3
29   ...
30 }
```

Listing 3: TRC for the AADL refinement alternatives

7.3 Extraction of Atomic Transformation Instantiations

From an input model, a TRC, and a set of ATL modules, the extraction of atomic transformation instantiations (ATIs) follows the following process:
1. the pattern matching of ATL rules is executed on the input model, and produces TRIs;
2. from these TRIs, concistency contraints expressed in the TRC are instantiated to produce validity constraints over the selection of TRIs (as described in section 5);

3. TRIs are partitionned using a conjunctive normal form of the validity constraints (as described in section 6);
4. using SAT techniques, sets of ATIs are produced for each partition of TRIs. The genetic algorithm presented in section 6 can then be implemented using ATIs as values in the genome encoding.

In this section, we have presented model transformation alternatives dedicated to the replication transformation. Similar artefacts were developed for the allocation of software components on hardware components. Software components replication and allocation transformations were then chained during the validation of our approach, which is presented in next section.

8 Approach Validation

In this section, we report the validation of the proposed approach as follows: in subsection 8.1, we enumerate and comment the research questions we answer. We then describe the experiments setup in subsection 8.2. In subsection 8.3, we present the experimental results and our conclusions with respect to research questions. Finally, subsection 8.4 discusses the threats to validity of these conclusions.

Note that the experiments presented in this section can be reproduced by using the VirtualBox virtual machine, available online[2].

8.1 Research questions

In order to evaluate the quality of the proposed approach, we need to answer to the following research questions (RQs):

RQ1: how efficient is it to proceed to structural constraints validation a priori instead of a posteriori? This research question aims at evaluating if, when using EAs to explore a design space made up of architectural candidates, it is more efficient to proceed to the validation of structural constraints **a priori** rather than **a posteriori**.

RQ2: what is the distance between results obtained with our approach and theoretical local optima? This research question aims at evaluating the intrinsic quality of architectural solutions found with our approach.

RQ3: how much resource (memory and computation time) does our approach require to explore numerous architectural candidates? This research question aims at evaluating the complexity of our approach, in terms of computation time and memory space.

8.2 Experiments setup

To answer these questions, we provide results obtained on 4 models with increasing complexity. The characteristics of these models are listed in table 1,

[2] `http://perso.telecom-paristech.fr/~borde/publications/2017/sosym/Ubuntu.zip`

allowing to estimate the complexity of the design space. For instance, *model 1* is made up of 3 paths, 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 1: 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 configured our method, based on NSGA-II as follows: the size of the population was set to 100, the maximum number of iteration was set to 10 000 (10 for the first transformation, 1000 for the second one), and the maximum number of iterations without finding new non-dominated solutions was set to 5.

8.3 Experimental results

In this section, we use a classical vocabulary to present well-designed experiments: the term **independent variable** correspond to a factor we control during the experiment in order to observe its effect on so called **dependent variables**, which are measured as an outcome of the experiments.

8.3.1 RQ1: A priori vs. a posteriori validation

Evaluation method. To answer RQ1, we compared 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. In this experiment, the **independent variable** is the activation (or deactivation) of the a priori validation of structural constraints. When a piori validation is activated, structural constraints are not validated a posteriori. When a piori validation is deactivated, structural constraints are validated a posteriori. The **dependent variable** is the convergence time of the method we propose in this paper (either with a priori or a posteriori validation).
Results. Out of a thousand candidate architecture generated by the approach

based on **a posteriori** validation, none of them was correct with respect to structural constraints. On the other hand, the approach based on a priori validation was able to converge by evaluating more than one thousand correct architectures in about 2 to 12 hours of computation (respectively for the simplest and most complicate models we considered). This shows the importance of validating composite transformations **a priori** rather than a posteriori.

8.3.2 RQ2: distance to local optima

Evaluation method. To answer RQ2, we compare the distance between solutions obtained with our approach and theoretical local optima. However, the absolute distance is not very meaningful: 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 find the best theoretical optimum, we consider a subset of the design space to explore, and implement a Mixed Integer Linear Program (MILP) formulation of the optimization problem. The worst theoretical optimum is found by reversing the MILP optimization problem: the best local optimum is found by maximizing an objective function, while the worst is found by minimizing the objective function. The MILP-based solution required several adjustments that were made manually. First, we had to linearize the function that computes reliability. To do so, we 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 $\rho^{'}$ must be allocated on the same processor, all the processes of $\rho^{''}$ must be allocated on the same processor, etc. Under these restrictive hypothesis, each design pattern is associated a constant reliability value in the MILP formulation. As a consequence, the MILP considers only a subset of the possible architectures. In addition, this method is not applicable if processors of different reliability are used: this situation was avoided in our experiments in order to be able to obtain comparative results. Second, the MILP formulation required to linearize the multi-objective problem defined in section 3 into a single objective function. We defined this objective function as a weighted sum of the following objectives: reduce response-time and increase reliability. In this experiment, the **independent variable** is the number of alternative ATIs, which influences to the combinatorial complexity of the design space to explore. We control this variable thanks to the four models presented in section 8.2. The **dependent variable** is relative distance between the best solution obtained with our method and a local optimum obtained with the MILP formulation.

Results. Comparing results obtained with MILP and our approach, we came 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 observed for the two first models that the distance was 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 observed that our method found solutions that are not dominated by solutions found in MILP. These solutions were typically better in terms of reliability than solutions found by the MILP. Compared to a random exploration, our approach converged after evaluating 1200 (considering *model 1*) to 2020 (considering *model 4*) output models. Random exploration did not converge and stopped after evaluating 5000 to 10 000 output models. No solution found with the random method dominated 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.

8.3.3 RQ3: resources consumption

Evaluation method. To evaluate the quantity of resources required by the execution of our method, we measured the peaks of memory space and computation power it was using at runtime, as well as the time it needed to converge. In this experiment, the **independent variable** is the number of alternative ATIs, which influences to the combinatorial complexity of the design space to explore. We control this variable thanks to the four models presented in section 8.2. The **dependent variables** are memory, CPU occupation, and time required for our method to converge.

Results. Table 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 an important increase in terms of resource consumption from *model 1* to *model 4*. Measuring the time spent in different parts of our implementation, we could identify that an important part of the computation time (more than 50%) was 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 [25]. Our framework indeed executes HOTs to transform transformation alternatives into a single transformation that implements the selection of ATIs. This solution was initially made not to modify the ATL runtime, but it appears to be very time consuming in practive. Consequently, we believe the execution time we measured would be significantly improved by re-implementing this part of our framework.

Results provided in table 2 show that the convergence time is very high. Nevertheless, the approach can be set up to provide intermediate results after a predefined execution time so that architects may understand how the execution of the method is performing.

Table 2: Measured resource consumption

model identifier	Convergence time	Evaluated architectures	memory usage
model 1	2h34m	1102	6.4 GB
model 2	4h33m	2020	6.4 GB
model 3	11h29m	1818	14 GB
model 4	12h40m	1465	14 GB

8.4 Threats to validity

In this section, we discuss several threats to validity of the results presented in this paper. We use a classical vocabulary to present well-designed experiments:

- internal validity refers to the evaluation of potential factors (other than dependent variables) that may have produced the observed results.
- construct validity aims at validating that the observed results correspond to expected results according to the theory.
- external validity evaluates the possibility to generalize the observed results.

Internal validity. With respect to our results in response to RQ1, we do not see any internal threat to validity. To the best of our understanding, the activation/deactivation of a priori validation is the only factor responsible for the results we observe. For our answer to RQ2, the configuration of NSGA-II might be an internal threat to validity. Indeed, the proximity of results obtained with our method and with MILP could be improved by modifying the parameters of NSGA-II. For our answer to RQ2, the configuration of NSGA-II might be an internal threat to validity. Indeed, the proximity of results obtained with our method and with MILP could be improved by modifying the parameters of NSGA-II. Consequently, it is possible that the results obtained with model 1 are better than results obtained with other models because the parameterization of NSGA-II we used is more suitable for the optimization problem entailed by model 1. Note that we used the same parameterization of NSGA-II for all the models involved in this experiment. We did experiment with other parameterizations but we observed the same results: when we increase number of alternative ATIs, our method produces results that get further from local optima. When it comes to our answer to research question RQ3, the main internal threat to validity comes from the possibility to optimize our framework. As mentioned in section 8.3, more than 50% of the CPU time was spent in HOTs which were meant to ease the implementation but eventually introduced unnecessary complexity at runtime. Besides, this complexity rapidly grew with the number of alternative ATIs in our experiments. In addition to the impact of the complexity of the design space induced by the number of ATIs, we may also observe here the impact of unnecessary HOTs.

Construct validity. To the best of our understanding, the results we observe in this experiment conform to the theoretical contributions presented in this paper: a priori validation was expected to have a huge impact on convergence time, and we also expected the complexity to grow rapidly with the number of alternative ATIs.

External validity. 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 design 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 a previous experiment [22].

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.

We also acknowledge that experimental results provided in this paper rely on a small set of input models. However, one of them was 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 used are replication patterns used in industry, and their specification using ATL model transformations is quite complete. It is thus difficult to generalize the results of this paper to other models and design alternatives, but the experimental results we provided already demonstrate the feasibility of our approach on significant design problems.

9 Related Works

This section, dedicated to position our contribution with respect to existing works, is organized around three topics: (i) composition of model transformations, (ii) multi-objective optimisation of component architectures, and (iii) design space exploration based on model transformations.

9.1 Model Transformations Composition

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 [26]. First, *external* composition techniques consist in chaining model transformations: the target of a model transformations becomes the source model of the next transformation of the chain. Second, *internal* composition techniques aim at producing a new model transformation by merging existing model transformations. *Higher Order Transformations (HOTs)* is a well established technique to implement internal composition. We used HOTs to produce composite transformations from existing alternatives, following the principles described in [18]. In this paper, we automate the

exploration of a design space made up of chains of alternative model transformations. To make this contribution possible, we encode composite transformations into genomes for applying EAs. In addition, we use satisfiability techniques to make sure the resulting transformations always produce architecture satisfying predefined structural constraints.

However, the genome encoding of a transformation chain is very difficult to determine. Indeed, the size of the design space to explore in each link of the chain depends on the content of the model produced by previous transformations of this chain. For this reason, we rely on both composition techniques in our contribution: for each link in a transformation chain, we use internal composition to merge transformation alternatives into composite transformation. We use external composition to chain these composite transformations and produce candidate architectural models.

Elaborating valid transformation chains from a repository of independent model transformations is a difficult task, even more with exogeneous model transformations [11,3,10,4,7,6]. For model transformation chains, we consider simplifying hypothesis: model transformations are endogeneous and chained in a predefined static order. Yet, our work is to the best of our knowledge the first proposal to use both external and internal composition techniques to produce model transformations during design space exploration.

9.2 Multi-Objective Architecture Optimization

Multi-objective architecture optimization is a well established research field [2]. Most research works in this field focused on specific optimization problems, aiming at improving EFPs of architectural models. These works usually focus on the efficiency of the optimization technique, and its tuning for a given optimization problem. Nevertheless, some research works in this field also aim at bridging the gap between architecture optimization techniques and MDE or CBSE techniques [1,19,17]. These works aim at providing more generic solutions, applicable for a wider set of optimization problems.

Aleti et al. [1] developed a framework called ArcheOpterix, using AADL to describe input architectures. This framework defines interfaces for architecture analysis, optimisation, and constraints validation. ArcheOpterix has been experimented with several multi-objective optimisation techniques including evolutionary algorithms, ant colonies, or simulated annealing.

Following the same principles, Martens et al. [19] developed a framework called PerOpteryx, using the Palladio Component Model to describe input architectures. This framework proposes to use EAs, combining genetic operators and so called tactics, in order to optimize architectures exhibiting several degrees of freedom. In PerOpterix, tactics encode the expertise of designers with well identified architectural patterns. Tactics in PerOpteryx are similar to model transformations in our approach. However, as opposed to model transformations formalized with a dedicated language, tactics are internal operators of PerOpteryx and their applicability is detected dynamically during

the optimization process. As a consequence, it is difficult to constrain *a priori* the composition of tactics.

Li et al.[17] developed a framework called AQOSA, using AADL to describe input architectures. This framework also integrates modelling technologies, performance analysis methods, and EAs to improve extra-functional properties of architectural models. In addition, they provide a comparison of results obtained with different versions of EAs.

Contrasting with these approaches, RAMSES relies on a rule-based model transformation language and brings the following benefits: first, explored architectures are guaranteed to respect predefined structural constraints **a priori**, *i.e.* before the transformation is executed. Second, design alternatives are more precisely defined using model transformations, and complex design spaces can be explored by chaining transformations. Contrasting with our work, existing approaches do not provide solutions when (i) several design steps have to be explored and (ii) EFPs of candidate architectures can be assessed only if design decisions have been made for each step. Last but not least, the exploration engine is generic, and can be completly reused to solve different types of optimization problems. Appart from the model transformation chains, our framework was already experimented on another case-study using the same exploration engine [22]. Of course, the genericity of our solution has a price in terms of efficiency to converge towards good or near-optimal architectures. However we demonstrated in our experiments that when enforcing structural constraints **a priori** we were able to find solutions; but considering these constraints **a posteriori** did not work. We thus believe that our framework, despite of its genericity, should be more efficient than existing approaches when architectures to explore have to respect stringent structural constraints. In addition, our experience shows structural constraints do appear rapidly when precisely defining design alternatives (using model transformations for instance).

In the next subsection, we study related works aiming at automating design space exploration by model transformations.

9.3 Design Space Exploration By Model Transformations

To deal with quality attributes of architectures using model transformations, a first approach was proposed in [20]: given a set of architectural patterns, and a catalogue of model transformations, a method was proposed to guide architects towards architectures that comply with EFPs. However, this method is limited to horizontal model transformations (*i.e.* model transformations with the same abstraction level for the source and target model). Insfran et al. [14] reused the notion of quality-driven model transformations in a semi-automatic approach to select architectural alternatives. Contrasting with our approach, this method requires to know *a priori* the impact of model transformations on EFPs of produced architectures. With complex architectures and model transformations, this knowledge might be difficult to formalize. Similarly, Loniewski

et al. [18] proposed a semi-automatic approach, based on HOTs, to compose model transformations in order to improves EFPs.

Besides, none of these approaches is fully automated. This becomes a strong limitation when considering huge design spaces. To overcome this limitation, Drago et al. [9] proposed an extension of the model transformation language *qvt-relations* in order to automate design space exploration from the description of model transformation alternatives. However, to the best of our knowledge, the exploration engine proposed in this work does not rely on any well-identified optimization technique. As a consequence, it is difficult to assess whether this approach scales for complex design space exploration problems.

10 Conclusion and Future Works

In this paper, we propose an approach combining multi-objective optimization techniques and model transformation techniques to explore a design space made up of architectural candidates. In this approach, design alternatives are specified using a well-known model transformation language, ATL, thus providing several advantages with respect to existing methods: the exploration engine can be reused withtout modifications for different optimization problems, EFPs analysis methods can be reused to evaluate candidate architectures (*i.e.* output models), the verification that explored architecture respect predefined structural constraints is efficiently performed using SAT solving techniques, and complex design spaces can be explored by chaining model transformations.

Facing the problems raised by architecture exploration, and in particular the combinatory explosion of transformation alternatives, we automated the composition of transformations using an evolutionary algorithm. This required to structure model transformation alternatives in order to guarantee that genetic operators (*e.g.* mutation, crossover) are correctly applied: with our solution, new transformation alternatives resulting from the application of genetic operators necessarily produce correct architectures (*i.e.* satisfying structural constraints).

Our method was prototyped in RAMSES, an AADL refinement framework that interleaves model transformations and analysis to automate the production of real-time embedded systems. Our approach was experimented on several source AADL models, and provided results showing the capacity of our method to explore efficiently the design space generated by transformation alternatives.

In our future works, we plan to improve the performances of our architecture exploration engine 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 HOTs.

References

1. Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: Archeopterix: An extendable tool for architecture optimization of aadl models. In: Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES '09, pp. 61–71. IEEE Computer Society, Washington, DC, USA (2009)
2. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: A systematic literature review. IEEE Trans. Softw. Eng. **39**(5), 658–683 (2013)
3. Aranega, V., Etien, A., Mosser, S.: Using Feature Model to Build Model Transformation Chains, pp. 562–578. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated Chaining of Model Transformations with Incompatible Metamodels, pp. 602–618. Springer International Publishing, Cham (2014)
5. Borde, E., Rahmoun, S., Cadoret, F., Pautet, L., Singhoff, F., Dissaux, P.: 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 (2014)
6. Castellanos, C., Borde, E., Pautet, L., Sbastien, G., Vergnaud, T.: Improving reusability of model transformations by automating their composition. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, pp. 267–274 (2015)
7. Chenouard, R., Jouault, F.: Automatically Discovering Hidden Transformation Chaining Constraints, pp. 92–106. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. Trans. Evol. Comp **6**(2), 182–197 (2002)
9. Drago, M.L., Ghezzi, C., Mirandola, R.: A quality driven extension to the qvt-relations transformation language. Comput. Sci. **30**(1), 1–20 (2015)
10. Etien, A., Aranega, V., Blanc, X., Paige, R.F.: Chaining model transformations. In: 1rst Workshop on the Analysis of Model Transformations, pp. 9–14. NY, USA (2012)
11. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining independent model transformations. In: Symposium on Applied Computing, SAC '10, pp. 2237–2243. ACM, NY, NY, USA (2010)
12. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
13. Heckel, R., Kuster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: A. Corradini, H. Ehrig, H.J. Kreowski, G. Rozenberg (eds.) Graph Transformation, *LNCS*, vol. 2505. Springer Berlin Heidelberg (2002)
14. Insfran, E., Gonzalez-Huerta, J., Abrahão, S.: Design guidelines for the development of quality-driven model transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems: Part II, MODELS'10, pp. 288–302. Springer-Verlag, Berlin, Heidelberg (2010)
15. Jouault, F., Kurtev, I.: Transforming models with atl. In: International Conference on Satellite Events at MoDELS'05, pp. 128–138. Springer-Verlag, Berlin, Heidelberg (2006)
16. Koziolek, A., Koziolek, H., Reussner, R.: Peropteryx: Automated application of tactics in multi-objective software architecture optimization. In: the ACM SIGSOFT Conference Quality of Software Architectures. ACM, NY, NY, USA (2011)
17. Li, R., Etemaadi, R., Emmerich, M.T.M., Chaudron, M.R.V.: An evolutionary multiobjective optimization approach to component-based software architecture design. In: Evolutionary Computation (CEC), 2011 IEEE Congress on, pp. 432–439 (2011)
18. Loniewski, G., Borde, E., Blouin, D., Insfran, E.: An automated approach for architectural model transformations. In: 22nd International Conference on Information Systems Development (ISD2013). Sevilla Spain (2013)
19. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: First Joint WOSP/SIPEW International Conference on Performance Engineering, pp. 105–116. ACM, New York, NY, USA (2010)
20. Merilinna, J.: A tool for quality-driven architecture model transformation. Phd thesis, VVT Technical Research Centre of Finland, Vuorimiehentie, Finland (2005)
21. OMG, O.M.G.: Flow latency analysis with the aadl (2012)

22. Rahmoun, S., Borde, E., Pautet, L.: Automatic selection and composition of model transformations alternatives using evolutionary algorithms. In: 2015 European Conference on Software Architecture Workshops, pp. 25:1–25:7. ACM, New York, NY, USA (2015)
23. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. Computers, IEEE Transactions on **39**(9), 1175–1185 (1990)
24. Srinivas, N., Deb, K.: Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation **2** (1994)
25. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: 5th European Conference on Model Driven Architecture - Foundations and Applications. Springer-Verlag, Berlin, Heidelberg (2009)
26. Wagelaar, D.: Composition Techniques for Rule-Based Model Transformation Languages, pp. 152–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
27. Zheng, W., Zhu, Q., Natale, M.D., Vincentelli, A.S.: 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)
28. Zhu, Q., Yang, Y., Scholte, E., Natale, M.D., Sangiovanni-Vincentelli, A.: Optimizing extensibility in hard real-time distributed systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2009, pp. 275–284. IEEE (2009)