Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science - Software Engineering 30.0 credits

AUTOMATED SYNTHESIS OF MODEL COMPARISON BENCHMARKS

Lorenzo Addazi
lai15004@student.mdh.se

Examiner: Jan Carlson
Mälardalen University, Västerås, Sweden

Supervisor: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

June 15, 2019

Success is stumbling
from failure to failure
with no loss of enthusiasm.

WINSTON CHURCHILL

Abstract

Model-driven engineering promotes the migration from code-centric to model-based software development. Systems consist of model collections integrating different concerns and perspectives, while semi-automated model transformations generate executable code combining the information from these. Increasing the abstraction level to models required appropriate management technologies supporting the various software development activities. Among these, model comparison represents one of the most challenging tasks and plays an essential role in various modelling activities. Its hardness led researchers to propose a multitude of approaches adopting different approximation strategies and exploiting specific knowledge of the involved models. However, almost no support is provided for their evaluation against specific scenarios and modelling practices. This thesis presents Benji, a framework for the automated generation of model comparison benchmarks. Given a set of differences and an initial model, users generate models resulting from the application of the first on the latter. Differences consist of preconditions, actions and postconditions expressed using a dedicated specification language. The generator converts benchmark specifications to design-space exploration problems and produces the final solutions along with a model-based description of their differences with respect to the initial model. A set of representative use cases is used to evaluate the framework against its design principles, which resemble the essential properties expected from model comparison benchmark generators.

Table of Contents

1 Introduction 5
1.1 Problem Formulation 5
1.2 Thesis Contribution 6
1.3 Thesis Outline 6

2 Research Methodology 7

3 Background 9
3.1 Model Comparison 9
3.1.1 Matching 9
3.1.2 Differencing 10
3.1.3 Evalution 11
3.2 Design Space Exploration 12
3.2.1 Model Generation 12
3.2.2 Model Adaptation 12
3.2.3 Model Transformation 13

4 *Benji* – A Model Comparison Benchmark Generator 14
4.1 Design Principles 14
4.2 Overall Architecture 15
4.3 Trace Representation 15
4.4 Benchmark Specification 17
4.4.1 Difference Specification Language 17
4.4.2 Benchmark Specification Language 18
4.5 Difference-Space Exploration 19
4.6 Output Construction 21

5 Evaluation 22
5.1 Simplified Ecore 22
5.2 Metamodel Refactorings Catalog 22
5.2.1 Push Down Attribute 24
5.3 Design Principles 25

6 Discussion 28

7 Related Works 29

8 Conclusion 31

References 34

List of Figures

1 Research Methodology . . . 7
2 Model Comparison – Overview . . . 9
3 Simple Family – Metamodel and Models . . . 10
4 Model Differencing – Operation-based vs. State-based . . . 10
5 Model Generation Pattern . . . 12
6 Design-Space Exploration – Model Adaptation Pattern . . . 12
7 Design-Space Exploration – Model Transformation Pattern . . . 13
8 *Benji* – Overall Architecture . . . 15
9 Trace Representation – Metamodel . . . 15
10 Trace Representation – Created Element . . . 16
11 Trace Representation – Deleted Element . . . 16
12 Trace Representation – Preserved Element . . . 16
13 Trace Representation – Changed Element . . . 16
14 Difference Specification Language – Metamodel . . . 17
15 Benchmark Specification Language – Metamodel . . . 18
16 *Benji* – Output Construction . . . 21
17 Push Down Attribute – Structural Description . . . 24
18 Push Down Attribute – Difference Model . . . 25
19 Completeness – Initial Model . . . 25
20 Completeness – Expected Difference Combinations . . . 26
21 Pseudo-Randomness – Initial Model . . . 26
22 Visibility – Initial Model . . . 27
23 Minimality – Initial Model . . . 27

Listings

1 Trace Representation - VQL Patterns . . . 16
2 Rename Person - Difference . . . 17
3 Rename Person - Precondition . . . 18
4 Rename Person - Action . . . 18
5 Rename Person - Postcondition . . . 18
6 Family Benchmark Specification . . . 19
7 Simplified Ecore Metamodel [1] . . . 22
8 Universal Quantifier – Example Pattern . . . 23
9 Push Down Attribute – Difference . . . 24
10 Push Down Attribute – Precondition . . . 24
11 Push Down Attribute – Action . . . 24
12 Push Down Attribute – Postcondition . . . 25
13 Completeness – Benchmark Specification . . . 26
14 Pseudo-Randomness – Benchmark Specification . . . 26
15 Visibility – Conflicting Differences . . . 27
16 Minimality – Benchmark Specification . . . 27

List of Tables

1 Comparison Results – Negatives and Positives . . . 11
2 Model Generation Frameworks – Design Principles Comparison . . . 19
3 Evaluation – Metamodel Refactorings Catalog . . . 23
4 Model Generation Frameworks – Design Principles Comparison . . . 30

1 Introduction

Model-driven engineering (MDE) is a software development methodology promoting the migration from a code-centric to a model-based approach. Traditional software engineering practices consider models as mere documentation artefacts whom main purpose consists in facilitating the communication among stakeholders, i.e. descriptive models [2]. The most important artefact of the development process consists of executable code, which is manually produced and maintained coherent with the other artefacts. In a model-driven approach, instead, the system under development is represented as a collection of models either focusing on different concerns or providing information about the existing relationships between distinct views, i.e. prescriptive models [2]. Models assume the role of first-class artefacts throughout the development process, whereas the executable code of the system is (semi-)automatically generated combining the information from these [3].

Increasing the abstraction level from source code to models required a review of the available tools and methods supporting the various software development activities. For this reason, model management and evolution techniques have been subject to intense research in the recent years [4]. Model comparison, i.e. the identification of the differences existing among models, represents one of the most challenging tasks and plays a crucial role in various modelling activities [5]. Detecting, analysing and understanding the correspondences between different versions of the same model is fundamental when evolving systems and exploring design alternatives [5, 6]. Model comparison also provide an essential support to maintain consistency among different views of a system when adopting a multi-view approach [7]. Moreover, performing model comparison represents the initial step of the model transformation development process [8] and could also be used in model transformation testing, i.e. comparing the produced model with the expected one [7, 9].

The comparison process is generally decomposed in two phases, i.e. matching and differencing [10]. Among these, the intrinsic complexity of model comparison results from the matching process [11]. Considering software models as typed graphs, indeed, determining the correspondences between model elements can be reduced to solving an instance of the graph isomorphism problem [7], which is known to be NP-Hard [12]. Therefore, the various methods proposed in the literature represent different attempts to provide an approximate solution exploiting structural [8, 13], language-specific [14, 15] or domain-specific [7, 16] knowledge of the involved models [6, 17].

> *"There is no single best solution to model matching but instead the problem should be treated by deciding on the best trade-off within the constraints imposed in the context, and for the particular task at stake."*
>
> – Kolovos et al. [11]

1.1 Problem Formulation

Researchers and practitioners need support to systematically evaluate model comparison methods with respect to specific application domains, languages or modelling practices. Unfortunately, the large number of model comparison approaches corresponds to the almost complete absence of support for their systematic evaluation. Despite being possible to find manually defined benchmarks, e.g. [18] and [19], these do not provide a plausible solution for every use case as modelling language, application domain, applied differences and their representation are fixed. Furthermore, the design rationale is often not accessible and potentially biased, which makes the results unreliable and irreproducible. Reusing existing models, instead, would still require the careful identification and representation of the differences existing among these. The assessment of model comparison algorithms thus becomes a time-consuming and error-prone task requiring combined knowledge about the modelling languages, application domain and comparison algorithms in analysis.

The main objective of this thesis consists in providing support to researchers and practitioners for the definition of domain-specific model comparison benchmarks through an automated procedure. In particular, the procedure relies on design-space exploration techniques inspired by its successful application to automate similar model management tasks [20]. Given a set of difference specifications and an input model, users are able to generate a specified number of mutants resulting from the application of the first to the latter. For each mutant model, a description of the

applied changes is also generated. In order to achieve the main objective, the research process is driven by the following research questions:

RQ1 What kind of information is required for the specification of differences between models?

RQ2 How can we formulate the creation of model comparison benchmarks as design-space exploration problem?

RQ3 How can we represent the applied differences supporting their adaptation according to the requirements of an arbitrary comparison algorithm in analysis without loss of information?

1.2 Thesis Contribution

Understanding what kind of information is involved in the specification of differences among models (RQ1) required researching the state-of-the-art in model comparison approaches with specific focus on their matching technique. The obtained findings served as basis for the design of an operational notation representing differences in terms of preconditions, actions and postconditions. Preconditions and postconditions consist of assertions describing properties of the involved model elements before and after their modification, respectively. Actions, instead, contain the actual imperative edit statements applied on the involved models.

Formulating the creation of model comparison benchmarks as design space exploration problem (RQ2) required gathering information related to design space exploration techniques in model-driven engineering. The following step consisted in developing a design space exploration formulation, i.e. input, output, objectives, constraints, state-coding. The remaining research question (RQ3) required reviewing model difference representation techniques focusing on their expressive power and support for subsequent model manipulations.

Integrating the obtained results produced a framework for the systematic generation of model comparison benchmarks – *Benji*. In order to evaluate the framework against its desired properties, an illustrative benchmark construction example has been designed and reproduced using a prototype implementation based on the *Eclipse Modelling Framework* [21] and *Viatra-DSE* [22].

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Section 3 introduces the reader to the fundamentals required for a clear understanding of the problem, its challenges and the solution proposed in this thesis, i.e. model comparison and design-space exploration in model-driven engineering. Section 2 illustrates the research methodology followed throughout the thesis process. Section 4 presents *Benji*, a framework for the generation of domain-specific model comparison benchmarks. The discussion is decomposed with respect to its input, output and execution. Section 5 presents a proof-of-concept implementation of the framework based on the *Eclipse Modelling Framework* [21] and *Viatra-DSE* [22]. The implementation is then evaluated against an illustrative example readapting well-known metamodel refactorings from [23] on a simplifed version of the *Ecore* metamodel [1]. Section 7 compares our contribution with related works in the literature either addressing the assessment of model comparison algorithms or generating models with user-defined properties. Section 8 concludes the thesis summarising the obtained results and possible future directions.

2 Research Methodology

The main objective of this thesis consisted in providing support to researchers and practitioners in the creation of model comparison benchmarks through a systematic procedure based on design space exploration. The procedure is designed to integrate information related to specific application domains, modelling languages or differences, while maintaining a clear separation of concerns enabling reusability and extensibility of the generated benchmarks. According to the classification proposed in [24], the main contribution of this thesis falls within the *"procedures and techniques"* category. The thesis work proceeded following an adapted implementation of the engineering design process, an iterative decision-making workflow supporting engineers in creating products [25], as illustrated in Figure 1.

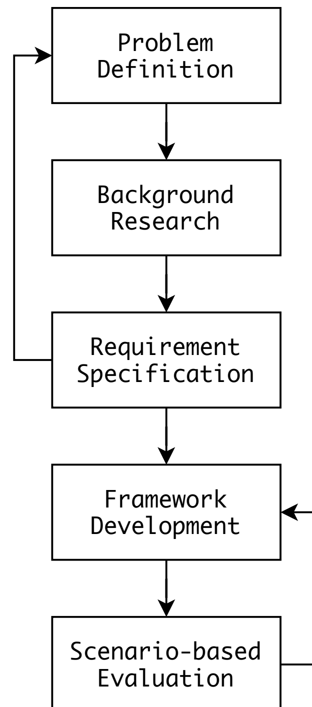

Figure 1: Research Methodology

The research process started defining the problem of interest. An initial and broad definition of the problem emerged from previous experiences in extending a model comparison method [16] and found confirmation after a quick investigation of the existing literature addressing the assessment of model comparison algorithms [26, 27]. During the problem definition phase, the main objective consisted in producing a clear definition of the problem and the research questions that needed to be answered – Section 1.

The research questions narrowed the thesis scope and acted as guidance during the background research phase. In this, contributions related to model comparison and design space exploration techniques in model-driven engineering have been researched using the snowball method [28] – Section 3.

The acquired knowledge supported the requirement specification phase, where expected functionalities and qualities for a systematic procedure supporting the creation of model comparison benchmarks have been defined. Using the specified requirements, the following step consisted in building a design space exploration, i.e. input, output, objectives, constraints, state-coding, and architecting a framework integrating the formulation while abstracting its technical details – Section 4.

An illustrative example has been designed considering functional and extra-functional requirements, and reproduced using a prototype implementation of the framework based on the *Eclipse Modelling Framework* [21] and *Viatra-DSE* [22]. The main objective consisted in demonstrating

the appropriateness of the framework, as well as analysing its limitations. Possible functional defects resulted in re-iterating the framework development activities.

Despite being common to validate procedures and techniques using examples [24], this approach introduces threats over the generalisability of the obtained results. In order to manage this risk, the illustrative scenarios have been designed exclusively considering the identified requirements, hence aiming to prove properties and functionalities not depending on the specific technologies and languages in use. As discussed in Section 8, evaluating the framework on large-scale industrial case studies is planned for the near future.

3 Background

This section introduces the essential concepts underlying the main contribution of this thesis. The discussion starts with an overview of model comparison, its composing activities, the different approaches proposed in the literature and common evaluation techniques. The section concludes introducing design-space exploration in model-driven engineering along with common integration patterns in the literature.

3.1 Model Comparison

Model-driven engineering promotes the migration from a code-centric to a model-based approach to cope with the increasing complexity of modern software systems development. Domain-specific modelling languages are defined using meta-models and used to create models focusing on different aspects of the system under development. Models assume the role of first-class citizens throughout the development process [3]. The information from multiple models is integrated into other artefacts and kept consistent using automated model transformations. Complex software system development is realised through chains of increasingly platform-specific transformations from high-level models to source code. In this context, disposing of efficient techniques detecting model-level differences represents a fundamental requirement for numerous activities spanning throughout the development process, e.g. model versioning [6], model transformation testing [9] and (meta)model co-evolution [7].

The existing approaches can be distinguished in two categories with respect to the number of models involved in the comparison process, i.e. two-way and n-way techniques. Intuitively, the first class groups all approaches limiting their comparison process to two models, i.e. initial and current, whereas the latter support comparisons involving an arbitrary number of models. The following discussion focuses on two-way model comparison approaches.

Given two models, model comparison algorithms produce a *difference model* illustrating the changes among these. As depicted in Figure 2, the comparison process is decomposed in two phases – *matching* and *differencing* [10]. Initially, all elements from the first are linked with the corresponding ones in the latter. Then, a difference model is constructed processing the identified correspondences.

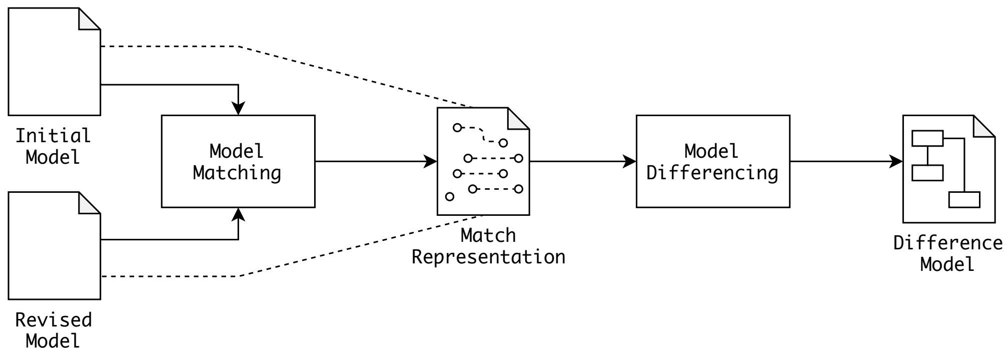

Figure 2: Model Comparison – Overview

3.1.1 Matching

Given two models, the matching phase consists in mapping the elements from the first with the corresponding ones in the latter. Each mapping might involve multiple elements from one model or the other, and provide a discrete or continuous numerical value indicating the plausibility of their correspondence. Considering their matching criteria, the existing approaches can be grouped in four categories – *identity-based*, *signature-based*, *similarity-based* and *language-specific* algorithms [11].

Identity-based and signature-based techniques associate model elements with persistent or dynamically generated unique identifiers, respectively. In this context, the matching criteria simply

consists in mapping elements having the same identifier and their similarity value is discrete, i.e. whether the related identifiers correspond or not.

Similarity-based matching algorithms compare model elements using dedicated similarity functions on their structural features, e.g. name, references or attributes. Unlike the previous approaches, mappings are associated with continuous values, hence elements from one model might correspond to multiple elements from the other, i.e. partial matches.

Language-specific algorithms extend the previous approaches exploiting semantic information concerning specific application domains or modelling languages to optimise their matching process.

3.1.2 Differencing

Given a match representation, model differencing algorithms translate the identified mappings into meaningful change descriptions. The existing techniques can be grouped in two categories – *operation-based* and *state-based*.

Operation-based notations represent changes in terms of edit primitives applied on the initial model, whereas state-based approaches construct declarative descriptions focusing on the visible changes in the final model. In other words, operational approaches describe *how* model elements changed, while state-based approaches focus on *what* changed.

In order to illustrate the difference between these approaches, let us consider the following small example metamodel and two conforming model instances – Figure 3. A simple *Family* metamodel is defined in the upper part, whereas the two different instances are represented in the bottom. Comparing these, the *name* attribute value of the *Family* instance *f1* is changed from *”family1”* to *”Doe”*, the *Person* instance *p1* is deleted and another *Person* instance *p2* is created with *name* attribute value equal to *”John”*.

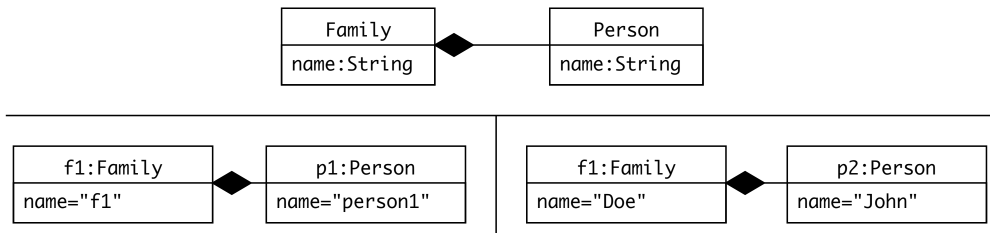

Figure 3: Simple Family – Metamodel and Models

Edit Scripts represent a well-known metamodel-independent operational approach involving primitives to create, delete and update elements – *new*, *delete*, *set*, *insert*, *remove* [5]. The notation presented in [29], instead, provides an example of state-based approach. Given a metamodel, an extended metamodel supporting the representation of modified model elements is automatically derived. For each metaclass *Class*, three corresponding metaclasses are generated representing created, deleted and changed instances, respectively – *ChangedClass*, *CreatedClass*, *DeletedClass*. The left and right side of Figure 4 illustrate the operation-based and state-based representation of the existing differences between left-side and right-side models in Figure 3, respectively.

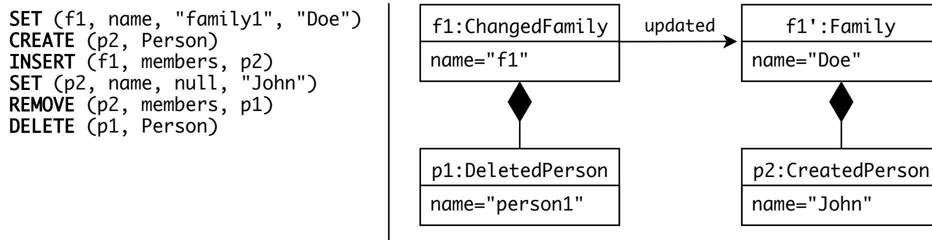

Figure 4: Model Differencing – Operation-based vs. State-based

3.1.3 Evalution

Given the essential role played within various model management activities, the concept of quality of model comparison algorithms is relative and the evaluation criteria used to assess their results depend on the specific use case [27]. For example, an algorithm producing low-level differences might result convenient in semi-automated workflows, e.g. model transformation testing. Inversely, the same differences could result hardly readable and inconvenient in tasks involving human interaction and reasoning, e.g. manual conflict resolution in model versioning.

Regardless of the evaluation criteria, model comparison algorithms are generally assessed constructing ad-hoc benchmarks consisting of triples of models $\langle M_1, M_2, \Delta_{M_1 \to M_2} \rangle$ where M_1 is the initial model, M_2 represents a possible modified version of M_1 and $\Delta_{M_1 \to M_2}$ acts as oracle describing the actual changes applied on M_1 in order to obtain M_2.

Given a model comparison benchmark, the expected differences and the actual results produced by an algorithm in analysis can be partitioned into four categories – *false negatives*, *true negatives*, *false negatives*, *true negatives* – as illustrated in Table 1. Negatives are differences that have not been identified during the comparison process, whereas positives consists in differences that have been identified. Both partitions are further decomposed in true and false depending on their correctness with respect to the expected differences. False positives, for example, represent identified differences that were not supposed to be detected, while true positives consist in expected differences that were successfully identified.

	Negatives		Positives	
	False	True	False	True
Identified	✗	✗	✓	✓
Expected	✓	✗	✗	✓

Table 1: Comparison Results – Negatives and Positives

Although an established definition of quality for model comparison algorithms still does not exist, the current evaluation approaches share a common concept of correctness concerning the produced results. In practice, the correctness of model comparison algorithms is quantified adapting fundamental metrics from the field of information retrieval on positives and negatives – *precision*, *recall* and *f-measure* (Equation 1–3). In particular, precision and recall compute the percentage of correct differences over all the proposed and the expected ones, respectively. Finally, f-measure combines precision and recall into a single harmonic mean value.

$$Precision = \frac{|\{True\ Positives\}|}{|\{True\ Positives\}| + |\{False\ Positives\}|} \tag{1}$$

$$Recall = \frac{|\{True\ Positives\}|}{|\{False\ Negatives\}| + |\{True\ Positives\}|} \tag{2}$$

$$F - Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

Complex and application-specific evaluation criteria might be defined over positives and negatives, as well as precision and recall. For example, the *overall* metric in [30] is defined to measure the effort required in order to align the results produced by a given comparison algorithm with the expected ones, i.e. adding false negatives and removing false positives. Intuitively, this metric aims to measure the effectiveness of a given model comparison algorithm within a model versioning workflow.

$$Overall = Recall \cdot (2 - \frac{1}{Precision}) \tag{4}$$

3.2 Design Space Exploration

Design-Space Exploration is an automated search process where multiple design alternatives satisfying a set of constraints are discovered and evaluated using goal functions. Numerous approaches exploiting design-space exploration techniques have been proposed and succesfully applied in different domains, e.g. model merging [31], software security [32], circuit design [33, 34], embedded systems development [35, 36]. In practice, three main classes of approaches integrating model-driven engineering with design-space exploration techniques have been identified – *Model Generation*, *Model Adaptation* and *Model Transformation* [37]. The main differences among these approaches regard the input artefacts, the adopted exploration techniques and the required expertise about the problem and the application domain.

3.2.1 Model Generation

The model generation pattern generates models conforming to an input metamodel while satisfying user-defined constraints expressed using a dedicated notation. The design-space exploration is reduced to solving a constraint satisfaction problem over models [38]. Figure 5 illustrates the pattern. Depending on the solver, metamodels and constraints are transformed using the expected problem formulation notation (A). The candidate solutions produced by the solver (B) are evaluated using user-defined goal functions and transformed into the initial or another modelling formalism (C). The exhaustive search performed in approaches implementing this pattern might result computationally inefficient. Therefore, initial metamodel instances might be used to impose additional constraints narrowing the search space from the beginning.

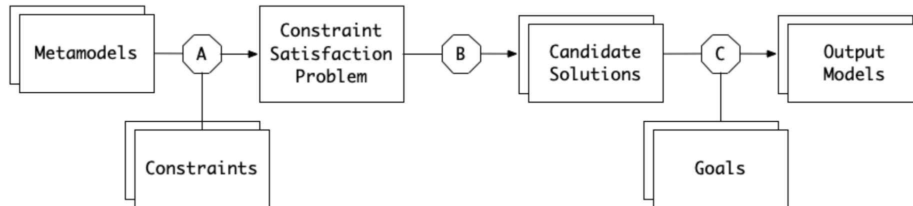

Figure 5: Model Generation Pattern

3.2.2 Model Adaptation

Approaches implementing the model adaptation pattern execute meta-heuristic search algorithms on ad-hoc search representations built from a set of initial models. Figure 6 illustrates the pattern. The specific search representation depends on the problem domain and search algorithm (A). During the exploration process, candidate solutions are generated applying user-defined manipulations on the initial search model. Global constraints prune the exploration process discarding invalid intermediate solutions, whereas goal functions are used to evaluate their optimality (B). The obtained solutions might be transformed conforming to the initial or another formalism (C). The user is required to provide model manipulations, global constraints and goal functions expressed using dedicated notations. This allows explicit integration of domain-specific knowledge.

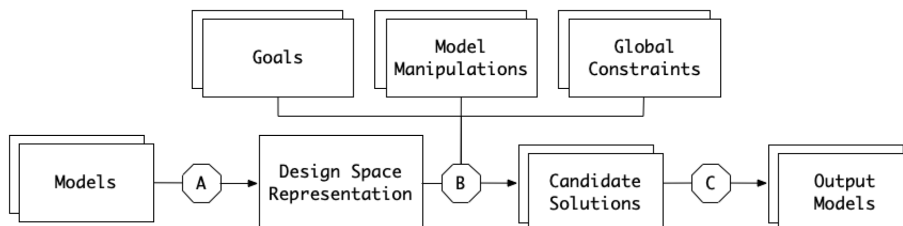

Figure 6: Design-Space Exploration – Model Adaptation Pattern

3.2.3 Model Transformation

Unlike the previous ones, the approaches implementing the model transformation pattern directly use the original input models throughout the design-space exploration process. Candidate solutions are obtained using user-defined model transformation rules, which selection criteria depends on the specific application. Figure 7 illustrates the pattern. Each candidate solution is validated against multiple global constraints along with the corresponding sequence of applied transformation rules (A). The output models are generally represented using the initial model notation and obtained selecting the candidate solutions optimising the result of the user-defined goal functions (B). The approaches implementing this pattern require deep knowledge about the specific problem and application domain. Furthermore, the efficiency and correctness of the exploration process strongly depends on the user-defined transformation rules and the trace representation formalism in use.

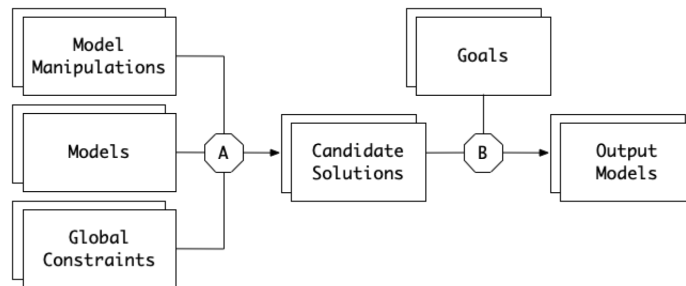

Figure 7: Design-Space Exploration – Model Transformation Pattern

4 *Benji* – A Model Comparison Benchmark Generator

This section presents the *Benji* model comparison benchmark generation framework, main contribution of this thesis. The framework consists of a domain-specific language for the specification of model comparison benchmarks and its interpreter. The discussion starts illustrating the design principles and overall architecture of the framework, then focusing on how models are represented and the features characterising the domain-specific language. Finally, the section concludes illustrating how the generation process is implemented using design-space exploration techniques and the output models are constructed starting from the obtained solutions.

4.1 Design Principles

In order to guide the development process, a set of fundamental properties concerning the systematic generation of model comparison benchmarks has been defined. These properties concern both the specification formalisms provided to the user, the generation process and the representation of the final results. *Configurability* has been defined reviewing the existing contributions proposing benchmarks for model comparison algorithms and extracting the essential configuration features that a benchmark generation framework must provide to users. *Completeness*, *Pseudo-Randomness*, *Minimality* and *Visibility*, instead, have been defined analysing the characteristics and limitations of the existing model generation approaches and languages. The first two principles represent recurring concerns in the existing literature on model generation techniques, as discussed in Section 7. The remaining principles, instead, have been defined observing the limitations of the existing model generation approaches. *Adaptability* has been defined investigating the various difference representation approaches used in the existing model comparison algorithms.

Configurability – Model comparison benchmarks are constructed aggregating triples, each consisting of an initial model, a modified version and a description of the differences among these. Therefore, the framework must allow to indicate the initial model and the differences to apply in order to obtain the modified versions. Furthermore, users must be able to indicate a minimum and maximum number of times each difference is expected to be applied, the number of modified models to generate and the location where to store these.

Completeness – Given a benchmark specification, the generation process must explore the complete solution space. More specifically, the framework must generate models considering all possible combinations of differences and their respective minimum and maximum number of expected applications.

Pseudo-Randomness – The users must be able to indicate the number of times a given difference is expected to be applied. However, it must not be possible to select the specific instance in case the same difference would be applicable multiple times. Throughout the generation process, the framework must mitigate the risk of introducing biases selecting the difference application to perform using a pseudo-random criteria.

Minimality – Duplicated models do not provide additional value to model comparison benchmarks. Therefore, the framework must ensure no duplicated models are generated. In particular, in case the same modified model could be obtained applying multiple sequence of difference applications, model comparison algorithms would select the shortest combination. Therefore, the framework must only keep the shortest sequence of difference applications for each duplicated model.

Visibility – Applying a given difference might might overwrite previous difference applications involving common model elements. Despite being applied, the previous difference would potentially be impossible to detect in the generated model. The framework must handle possible overlapping and conflicting differences throughout the generation process.

Adaptability – The generated models must be adaptable to evaluate model comparison algorithms using different notations and data granularity to represent their results. The applied

differences must be represented using a low-level and model-based notation allowing its conversion to algorithm-specific notations using semi-automated transformations and without requiring external information to be integrated.

4.2 Overall Architecture

The overall architecture of the framework is illustrated in Figure 8. Model comparison benchmarks are specified in terms of initial models and expected differences using a dedicated domain-specific language. Given a benchmark specification, the generation process can be decomposed in two phases – (A) *difference-space exploration* and (B) *output construction*.

During the difference-space exploration phase, the information from a given benchmark specification is processed and used to build and solve a corresponding design-space exploration problem instance. Further details on the formulation construction process are provided in Section 4.5.

Once concluded the exploration process, the obtained solutions are translated into the corresponding pair of output model and difference representation. Aggregating the generated pairs with the initial model produces a model comparison benchmark complying with the initial specification. The output construction process is further detailed in Section 4.6.

The *Benji* framework implements the model adaptation pattern presented in Section 3.2.2. Goal functions, model manipulations and global constraints are extracted from benchmark specifications during the difference-space exploration phase, whereas candidate solutions are transformed into output models during the output construction phase.

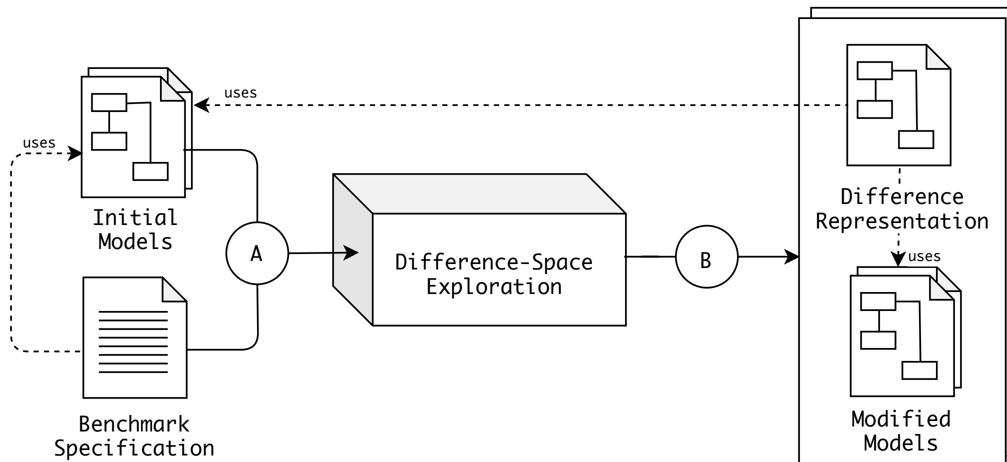

Figure 8: *Benji* – Overall Architecture

The framework is designed targeting the Eclipse Modelling Framework (EMF), a de-facto standard modelling platform for both academic and industrial projects [21]. However, its fundamental concepts and mechanisms abstract from implementation details and specific technological choices.

4.3 Trace Representation

Manipulating and reasoning about changing model elements throughout the generation process requires an appropriate formalism providing access to their current state while maintaining a read-only representation of their initial state. In *Benji*, such notation is implemented wrapping model elements into trace objects and splitting their representation into an initial and a current version as illustrated in Figure 9.

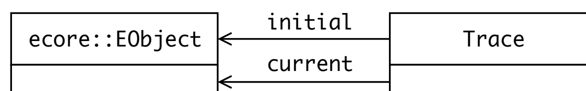

Figure 9: Trace Representation – Metamodel

Three possible states can be identified for model elements – created, deleted and preserved. In the first case, the corresponding trace representation consists of a trace object linked to the current model element version. The initial version, instead, is unset to indicate that the model element does not exist in the initial model, as illustrated in Figure 10.

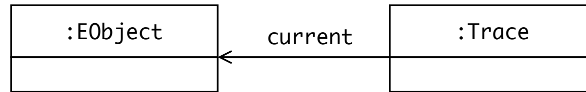

Figure 10: Trace Representation – Created Element

Similarly, deleted model elements are represented as trace objects linked to the initial model element version. The current version, instead, is unset indicating that the model element does not exist in the current model version, as illustrated in Figure 11.

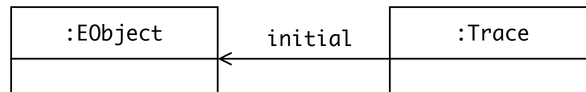

Figure 11: Trace Representation – Deleted Element

Finally, preserved model elements consist in elements that existed in the initial model version and continue to exist in the current version. Intuitively, the corresponding trace representation consists of a trace object linked to an initial and current model element instance, as illustrated in Figure 12.

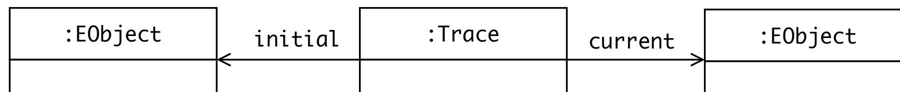

Figure 12: Trace Representation – Preserved Element

Edit operations are directly applied on the current model element version. Therefore, changes involving attributes and references are represented comparing initial and current version of a given model element. Figure 13 provides an example of renamed model element, i.e. where the name attribute value has been changed.

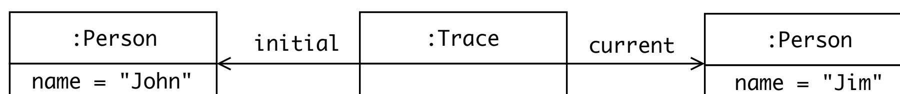

Figure 13: Trace Representation – Changed Element

In the current framework implementation, model element properties in preconditions and post-conditions are expressed using the Viatra Query Language (VQL). The patterns describing created, deleted and preserved model elements using their trace representation are illustrated in Figure 1.

In these, *trace_initial* and *trace_current* identify the initial and current model element version links, respectively. The *find* operator is used to check a given assertion and *neg* to represent negated conditions.

```
/* Created - element only existing in the current model */
pattern created (trace , current) {
    neg find trace_initial (trace , _initial);
    find trace_current (trace , current);
}
/* Deleted - element only existing in the initial model */
pattern deleted (trace : Trace , initial : EObject) {
  find trace_initial (trace , initial);
  neg find trace_current (trace , _current);
}
```

```
/* Preserved - element existing in both initial and current model */
pattern preserved (trace : Trace, initial : EObject, current : EObject) {
  find trace_initial (trace, initial);
  find trace_current (trace, current);
}
```

Listing 1: Trace Representation - VQL Patterns

4.4 Benchmark Specification

In *Benji*, two domain-specific languages are provided to define model comparison benchmarks and differences, respectively. These languages are built on top of existing technologies and designed to fulfil the configurability and visibility properties discussed in Section 4.1. The remainder of this section provides a description of the languages, while also illustrating their current implementation.

4.4.1 Difference Specification Language

Fulfilling the visibility principle requires the framework to provide means to express differences before and after their application, in addition to the actual modifications to perform. In this context, the visibility of a given difference can be verified ensuring that the corresponding post-condition is fulfilled in the current model. In order to provide this support, a dedicated difference specification language is defined in *Benji*. Figure 14 illustrates the main concepts of the language. Model differences are represented in terms of actions, preconditions and postconditions. An action consists of imperative statements representing the actual modifications to perform whenever the corresponding difference is applied, whereas precondition and postcondition consist of assertions describing the involved model elements before and after the difference application, respectively.

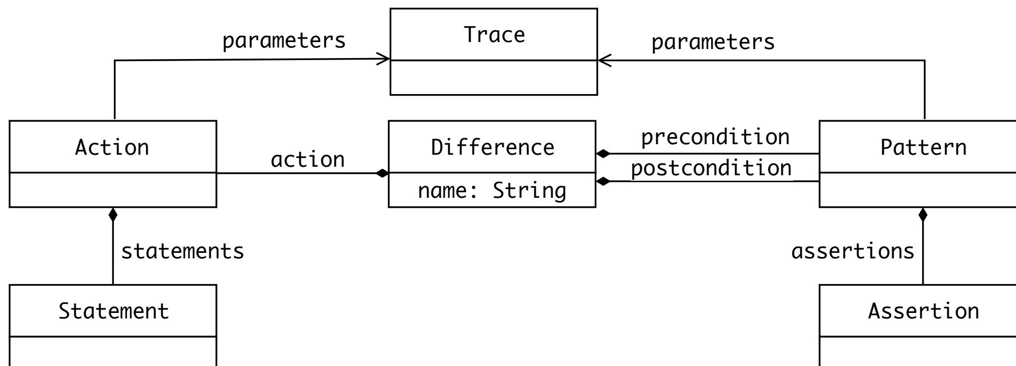

Figure 14: Difference Specification Language – Metamodel

The difference specification language is implemented as embedded domain-specific language in Xtend, a flexible and expressive dialect of Java [39]. Preconditions and postconditions are expressed using the Viatra Query Language (VQL), a domain-specific language for the specification of patterns over EMF models [40].

As illustrative example, let us consider the difference renaming a Person instance conforming to the Family metamodel presented in Section 3.1, as illustrated in Listing 2. Each difference specification starts with a unique *name* attribute, whereas *precondition*, *action* and *postcondition* are specified referencing the corresponding external definitions.

```
let renamePerson = difference
    .name("renamePerson")
    .precondition(beforeRenamePerson)
    .action(doRenamePerson)
    .postcondition(afterRenamePerson)
.build
```

Listing 2: Rename Person - Difference

Actions, preconditions and postconditions of the *renamePerson* example difference specification can be defined using trace objects, as illustrated in Listing 3–4. The difference involves a single *Person* instance, represented by the *person* parameter in precondition, postcondition and action. Intuitively, the precondition consists in finding a preserved *person* instance, which name property has not been changed before, i.e. initial and current value correspond (Listing 3). Given a *person* instance fulfilling the precondition, the action updates its current name attribute value, e.g. prepending a constant string to its initial value (Listing 4). Once applied the action, the postcondition still expects the involved *person* instance to be preserved, while also requiring its current name to be different from the initial (Listing 5).

```
pattern beforeRenamePerson (person : Trace) {
    find preserved (person, initial_person, current_person);
    find person_name (initial_person, initial_name);
    find person_name (current_person, current_name);
    initial_name == current_name;
}
```

Listing 3: Rename Person - Precondition

```
let doRenamePerson = [Trace person |
    person.current.name = "changed" + person.initial.name
]
```

Listing 4: Rename Person - Action

```
pattern afterRenamePerson (person : Trace) {
  find preserved (person, initial_person, current_person);
  find person_name (initial_person, initial_name);
  find person_name (current_person, current_name);
  initial_name != current_name;
}
```

Listing 5: Rename Person - Postcondition

4.4.2 Benchmark Specification Language

Fulfilling the configurability principle requires the framework to provide support for the specification of model comparison benchmarks in terms of initial models and differences with minimum and maximum number of expected applications. Furthermore, users must be able to specify the maximum number of generated models and the location where to store these. In order to provide this support, a dedicated benchmark specification language is defined in *Benji*. Figure 14 illustrates the main concepts of the language. Benchmark instances represent model comparison benchmark specifications and consist of an initial model and a set of bounded differences. The initial model is referenced using its resource location, whereas bounded differences refer to an existing difference specification and allows to specify a lower and upper bound on the number of expected applications. Finally, benchmark instances also provide support to specify the maximum number of generated models and their storage location.

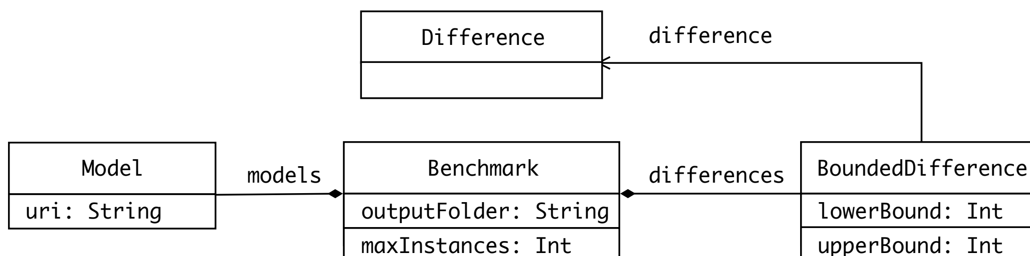

Figure 15: Benchmark Specification Language – Metamodel

The benchmark specification language is implemented as embedded domain-specific language in Xtend. Listing 6 provides an example benchmark specification in *Benji*.

```
benchmark
  .model("path/to/initial/model")
  .difference(0, 2, renamePerson)
  .difference(1, 1, deletePerson)
  .difference(0, 1, createPerson)
.build.generate(ALL, "path/to/output/folder")
```

Listing 6: Family Benchmark Specification

The benchmark specification integrates three differences, i.e. *renamePerson*, *deletePerson* and *createPerson*. The first corresponds to the difference illustrated in the previous section, whereas the remaining ones describe the creation and deletion of *Person* instances into *Family* instances, respectively. Each difference is associated with minimum and maximum number of expected applications, e.g. *renamePerson* is optional and can be applied two times at most for each generated model. The expected number of generated models can be specified as integer or left unbound with *ALL*, a special value used to generate all possible combinations of difference applications.

4.5 Difference-Space Exploration

The most important phase of the framework consists in generating models conforming to a given benchmark specification. The generation process essentially consists in searching for those modified versions of the initial model containing the expected differences. In *Benji*, this phase is implemented constructing and solving a design-space exploration problem instance. The choice of formulating the process of generating model comparison benchmarks as design-space exploration problem is based upon the successful application of these techniques to automate similar tasks requiring the search of models with specific characteristics, i.e. model transformation testing [41]. The formulation itself is part of the contribution of this thesis.

The mapping of benchmark specifications into design-space exploration problem instances has been designed considering the existing patterns presented in Section 3.2. More specifically, each pattern has been evaluated focusing on the support provided to fulfil the design principles concerning the exploration process defined in Section 4.1, namely *Configurability*, *Completeness*, *Pseudo-Randomness*, *Minimality*. The mapping among design-space exploration patterns and design principles is illustrated in Table 2.

	Model Generation	Model Transformation	Model Adaptation
Configurability	✗	✓	✓
Completeness	✗	✓	✓
Pseudo-Randomness	✗	✓	✓
Minimality	✗	✓	✓
Visibility	✗	✗	✓
Adaptability	✓	✓	✓

Table 2: Model Generation Frameworks – Design Principles Comparison

The model generation pattern neither does allow to specify an initial model nor the manipulations used to drive the exploration process and possible constraints that the generated models are required to fulfil. Consequently, no mechanism is provided to construct formulations conforming to the completeness and visibility design principles. Differently, both model transformation and model adaptation patterns construct their exploration process starting from an initial model. Furthermore, goal functions can be used to define constraints that the final solutions are required to satisfy, whereas global constraints can be used to impose conditions over all models throughout the generation process. Finally, both patterns adopt user-defined model manipulations to drive the exploration process. The main difference among these patterns consists in the model representation used to perform the exploration process. The first directly uses the initial model, while a dedicated search representation is used in the latter. In *Benji*, the initial model is transformed into a corresponding trace representation enabling the representation of changes over time. Consequently, the model adaptation pattern represents the most suitable solution. All three patterns support the adaptability design principles representing candidate solutions and output models using different formalisms. It is worth noticing that the completeness, pseudo-randomness and minimality design

principles depend on the strategy adopted for searching and selecting solutions, respectively. In this context, patterns are classified as supporting these principles if providing the possibility to choose the particular exploration strategy, e.g. depth-first or breadth-first. The model generation pattern fails to provide such support, whereas the remaining patterns allow to adopt different search strategies and impose dedicated goal functions handling the selection of minimal solutions only.

The mapping of model comparison benchmark specifications to design-space exploration problem instances is implemented constructing a constraint-satisfaction problem over models (CSP-M) [38] consisting of *initial models*, determining the initial exploration space, *goals*, distinguishing final from intermediate solutions, *global constraints*, required to be satisfied throughout the exploration process, and *model manipulations*, representing the available operations to manipulate one state into another. Once solved the problem instance, each solution consists in ordered sequences of model manipulations, i.e. *solution trajectories*. Given a benchmark specification, an equivalent design-space exploration problem is constructed as follows:

Initial Models – The initial model is transformed into an equivalent trace representation. Each model element is split into an initial and current version linked using a trace instance. Intuitively, the initial and current version of a given model element are completely equal at the beginning of the exploration process.

Goals – During the exploration process, lower and upper bounds associated with each differences are enforced using a dedicated goal over the solution trajectories. Given a solution trajectory T and a difference D, the goal verifies that the number of applications for D in T falls within its lower and upper bound values

$$LB(D) \leq CNT(D, T) \leq UB(D) \qquad \textbf{(Bounds Goal)}$$

where $CNT(D, T)$ computes the occurrencies of difference D within solution trajectory T, $LB(D)$ retrieves the lower bound associated with D and $UB(D)$ retrieves the upper bound associated with D in the benchmark specification. Considering the main objective of constructing a model comparison benchmark, generating models not containing any difference with respect to the initial model must be avoided. In other words, all solution trajectories must contain at least one difference application. Finally, given that each difference is associated with an upper bound over the number of expected applications in the specification. The maximum number of difference applications for each solution corresponds to the sum of all these upper bounds. Given a trajectory T, the length of solution trajectories is bound as follows

$$1 \leq LEN(T) \leq UB^{+} \qquad \textbf{(Length Goal)}$$

where UB^{+} is the sum of all upper bounds for each difference in the benchmark specification.

Global Constraints – Representing differences in three correlated portions, i.e. precondition, action and postcondition, is not enough to satisfy the visibility design principle. In addition to providing the possibility to express the consequences of a given difference, hence its visibility requirements, the framework must check and ensure their satisfaction throughout the exploration process. Global constraints are verified over all intermediate and final solutions, hence represent the ideal mechanism. Given a solution trajectory T, the global constraint verifies that the postcondition is fulfilled for each applied difference D_i composing the trajectory.

$$POST(D),\ \forall D_i \in T = \{D_0, ..., D_n\} \qquad \textbf{(Visibility Constraint)}$$

where $POST(D)$ verifies that the postcondition for difference D is satisfied in the current model, hence returns $true$ is positive and $false$ otherwise.

Model Manipulations – The available model manipulations driving the exploration process are extracted from the differences listed in a given benchmark specification. More specifically, given a difference specification, model transformation rules are composed using preconditions as guards and actions as bodies.

The difference-space exploration phase is implemented using *Viatra-DSE*, a design-space exploration framework for EMF models [42]. More specifically, each construct of the benchmark and difference specification languages is mapped to configuration steps to construct a design-space exploration problem. The languages act as configuration bridges similarly to the *Builder* design pattern [43].

4.6 Output Construction

Given the solution trajectories resulting from the difference-space exploration, the output construction phase handles the final step of the benchmark generation process. Each trajectory is applied on the initial models to generate the final ones, while an operation recorder keeps track of the applied changes and constructs the corresponding difference representation as illustrated in Figure 16.

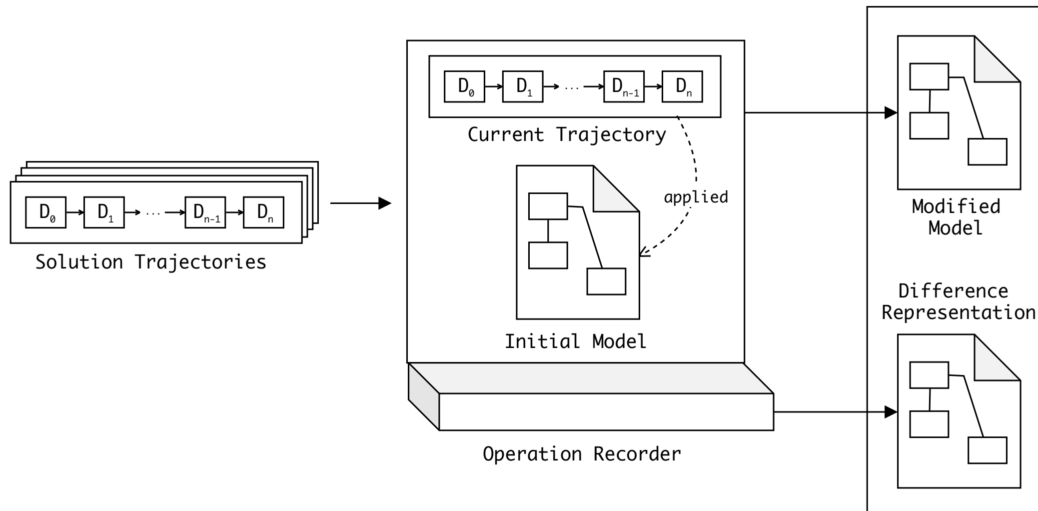

Figure 16: *Benji* – Output Construction

The operation recorder is currently implemented using the *EMF.Edit*, the standard EMF change monitoring support utilities [21]. The applied differences are represented using the state-based formalism introduced in Section 3.1.2. Considering the design principles discussed in Section 4.1, the framework fulfils the adaptability property providing low-level and model-based difference representations.

5 Evaluation

In order to evaluate the framework proposed in this thesis, various model comparison benchmark generation use cases have been designed. The main objective consisted in stressing out the expressivity of the domain-specific languages composing the framework, while confirming the correctness of both generation process and its outcomes. Furthermore, the evaluation served as reference to observe the appropriateness of the design principles underlying the framework, as well as its actual conformance to these during the development. The evaluation process consisted in developing differences resembling the well-known refactoring patterns listed in the *Metamodel Refactorings Catalog*, construct compatible input models and benchmark specifications, finally evaluating the generation process outcome. The expressive power of the languages composing the proposed framework has been evaluated attempting to implement the whole catalog, whereas benchmark specifications containing conflicting differences or limiting the number of models to generate have been designed to evaluate the framework against its design principles.

5.1 Simplified Ecore

Throughout the evaluation process, models have been constructed using the *Simplified Ecore* metamodel illustrated in Listing 7. All models are represented as root *Package* instances containing the other model elements. Packages are uniquely identified by their *Universal Resource Identifier*(URIs) attribute value and might contain three possible types of instances: *Package*, *Class* and *DataType*. Classes represent the core modelling concept of the metamodel. Each instance contains *Attribute* and *Reference* instances, possibly inheriting these extending other classes. Both references and attributes are typed with *DataType* and *Class* instances, respectively. References might be unidirectional or bidirectional, in which case an opposite reference can be defined. Finally, all elements provide a name attribute extending the *NamedElement* class.

```
class NamedElement {
  String name
}
class Package extends NamedElement {
  String uri
  contains Package[0..*] subPackages
  contains Class[0..*] classes
  contains DataType[0..*] dataTypes
}
class Class extends NamedElement {
  Boolean abstract
  refers Class[0..*] super
  contains Attribute[0..*] attributes
  contains Reference[0..*] references
}
class Attribute extends NamedElement {
  refers DataType type
}
class Reference extends NamedElement {
  refers Class type
  refers Reference opposite
}
class DataType extends NamedElement {}
```

Listing 7: Simplified Ecore Metamodel [1]

5.2 Metamodel Refactorings Catalog

In the *Metamodel Refactoring Catalog*, refactoring operations are classified with respect to three aspects – granularity, operation type and involved elements. The granularity indicates the nature of the transformation and can be atomic or composite. The first encompasses refactorings consisting of single editing operations, whereas the latter consist of multi-step operations. The operation type describes the kind of operations involved in the refactoring, i.e. add, delete or change. Finally, the involved model elements concerns the type of both the modified elements and their context, e.g. their content or container elements. Each refactoring pattern is described in

terms of motivation, possible usage example, actual modifications performed and structural representation of the involved model elements before and after its application. Table 3 illustrates the complete refactoring catalog along with information regarding whether or not the refactoring has been successfully implemented in the evaluation.

Name	Granularity	Operation Types	Implementation
Rename Package	Atomic	Change	✓
Rename Uri Package	Atomic	Change	✓
Delete Package	Composite	Delete	✓
Add Package	Atomic	Add	✓
Add Class	Atomic	Add	✓
Rename Class	Atomic	Change	✓
Delete Class	Composite	Delete	✓
Extract Class	Composite	Change, Add	✓
Merge Classes	Composite	Add, Delete, Change	✓*
Add Attribute	Atomic	Add	✓
Delete Attribute	Atomic	Delete	✓
Change Attribute Type	Atomic	Change	✓
Add Reference	Atomic	Add	✓
Delete Reference	Atomic	Delete	✓
Split References	Composite	Add, Delete	✓*
Merge References	Composite	Add, Delete, Change	✓*
Change Reference Type	Atomic	Change	✓
Extract Superclass	Composite	Add, Delete	✓
Change Class Abstract	Atomic	Change	✓
Restrict Reference	Atomic	Change	✓
Flatten Hierarchy	Composite	Add, Delete, Change	✓*
Push Down Attribute	Composite	Delete, Add	✓

Table 3: Evaluation – Metamodel Refactorings Catalog

The complete refactoring catalog has been successfully implemented using *Benji* in order to evaluate the expressive power of the proposed domain-specific languages, as illustrated in Table 3. Among the implemented patterns, those in the table that are marked with an asterisk required more complex reasoning in their precondition and postcondition. These limitations can be related to the existence of universal quantifiers over properties of the involved model elements and their non trivial implementation in terms of model constraint patterns, i.e. constraints over multiple instances. Universally quantified constraints can be defined comparing the number of elements matching a given property with the number of all elements. However, considering the expression *"All classes contained in the package contain all attributes matching property P."* and its corresponding pattern translation in Listing 8, it is possible to notice how multiple nested conditions might correspond to complex and verbose patterns, hence increasing the risk of errors.

```
pattern example (package) {
  package_classes == count find package_class (package, _class);
  package_classes_with_p == count find package_class_with_p (package, _class);
  package_classes == package_classes_with_p;
}

pattern package_class_with_p (package, class) {
  find package_class (package, class);
  class_attributes == count find class_attribute (class, _attribute);
  class_attributes_with_p == count find class_attribute_with_p (class, _attribute);
  class_attributes == class_attributes_with_p;
}
```

Listing 8: Universal Quantifier – Example Pattern

5.2.1 Push Down Attribute

As illustrative example, the following paragraphs describe the difference implementation corresponding to the *Push Down Attribute* refactoring. Moving properties represents a common refactoring operation whenever constructing class hierarchies. Specific properties might be aggregated into subclasses to decrease complexity, as well as pulled up to superclasses if representing common characteristics for a wider range of subclasses. The *Push Down Attribute* refactoring pattern describes the operation of moving a given attribute from a superclass down to its subclass. The actual modification composing this refactoring pattern consist in removing the selected attribute from the superclass and inserting this into the subclass, as illustrated in Figure 17. The difference specification simply consists of precondition, action and postcondition as illustrated in Listing 9.

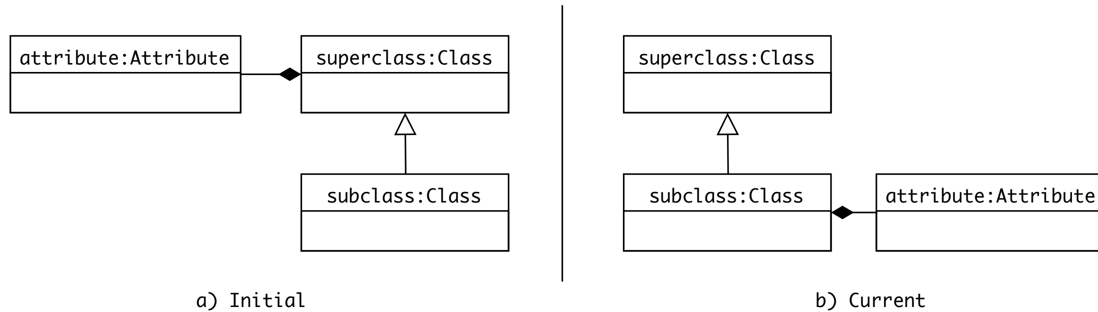

Figure 17: Push Down Attribute – Structural Description

```
let pushDownProperty = difference
  .name("pushDownProperty")
  .precondition(beforePushDownProperty)
  .action(doPushDownProperty)
  .postcondition(afterPushDownProperty)
.build
```

Listing 9: Push Down Attribute – Difference

The *Push Down Attribute* refactoring pattern involves class instances extending a superclass with at least one attribute. Consequently, the difference precondition expects three parameters – *class*, *super* and *attribute*. All parameter elements must be preserved, hence existing in both initial and current model. The *super* class must contain *attribute* and be a superclass for *class*. The corresponding VQL pattern implementation is illustrated in Listing 10.

```
pattern beforePushDownProperty (super, class, attribute) {
  find preserved (super, initial_super, current_super);
  find preserved (class, initial_class, current_class);
  find class_super (initial_class, initial_super);
  find class_super (current_class, current_super);
  find preserved (attribute, initial_attribute, current_attribute);
  find class_attribute (initial_super, initial_attribute);
  find class_attribute (current_super, current_attribute);
}
```

Listing 10: Push Down Attribute – Precondition

The actual modifications composing the refactoring consist in moving the attribute from the superclass to its subclass, hence one removal from the superclass and one insertion into the subclass. The corresponding action implementation is illustrated in Listing 11.

```
let doPushDownProperty = [super class attribute |
  super.current.attributes -= attribute.current
  class.current.attributes += attribute.current
]
```

Listing 11: Push Down Attribute – Action

Once applied the refactoring, the attribute is expected to be moved from the superclass into its subclass. In particular, the superclass is expected to contain the attribute in the initial model, while

the subclass is expected to contain the same attribute in the current model. The postcondition pattern implementation is illustrated in Listing 12.

```
pattern afterPushDownProperty(super, class, attribute) {
  find preserved (super, initial_super, current_super);
  find preserved (class, initial_class, current_class);
  find class_super (initial_class, initial_super);
  find class_super (current_class, current_super);
  find preserved (attribute, initial_attribute, current_attribute);
  find class_attribute (initial_super, initial_attribute);
  find class_attribute (current_class, current_attribute);
}
```

Listing 12: Push Down Attribute – Postcondition

Considering the initial model in Figure 17, the difference model describing the application of the *Push Down Attribute* difference on *superclass*, *subclass* and *attribute* is illustrated below. In this, *ChangedClass* and *ChangedAttribute* represent changed instances of the corresponding metaclass and are automatically generated from the metamodel.

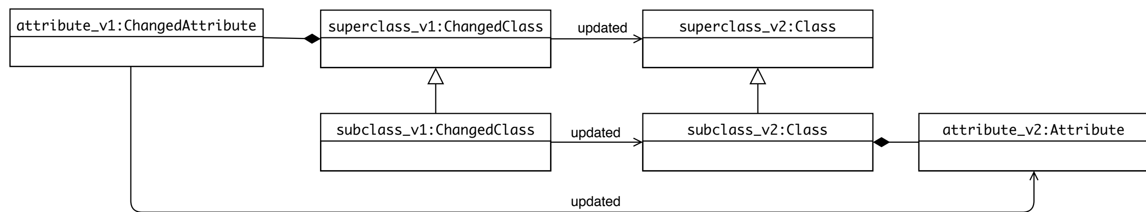

Figure 18: Push Down Attribute – Difference Model

5.3 Design Principles

The evaluation of the proposed framework against the design principles discussed in Section 4.1 has been conducted constructing various benchmark specifications focusing on eliciting each specific principle concerning the generation process, i.e. completeness, pseudo-randomness, minimality and visibility. In the following paragraphs, the evaluations are illustrated and discussed.

Completeness

Evaluating the framework against the completeness design principle requires veryfing that models resulting from all difference combinations according to their cardinalities are generated. Furthermore, the same results must be produced if no limit on the number of models is provided. The initial model consists of two *Class* instances. One class extends the other, which in turn contains an *Attribute* instance. Figure 19 illustrates the initial model.

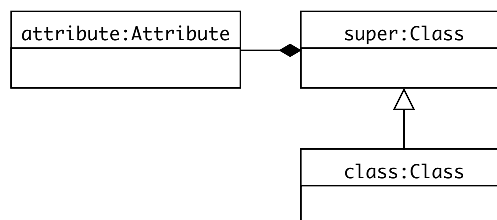

Figure 19: Completeness – Initial Model

The benchmark specification is illustrated in Listing 13 and includes two optional differences, i.e. *Push Down Attribute* and *Rename Class*. No limit is provided on the number of models to generate and no conflict exists among the differences.

```
benchmark
  .model("./resources/input/Input.xmi")
  .difference(0, 2, renameClass)
  .difference(0, 1, pushDownAttribute)
.build.generate(ALL, "./resources/output")
```

Listing 13: Completeness – Benchmark Specification

The expected difference combinations are listed in Figure 5.3, i.e. all distinct models resulting from all non-empty difference combinations. The framework produced the expected results and required 1,2 seconds to complete the generation process.

$$\{renameClass(class)\},$$
$$\{renameClass(super)\},$$
$$\{pushDownAttribute(super, class, attribute)\},$$
$$\{renameClass(class), renameClass(super)\},$$
$$\{renameClass(class), pushDownAttribute(super, class, attribute)\},$$
$$\{renameClass(super), pushDownAttribute(super, class, attribute)\},$$
$$\{renameClass(class), renameClass(super), pushDownAttribute(super, class, attribute)\}$$

Figure 20: Completeness – Expected Difference Combinations

Pseudo-Randomness

Evaluating the framework against the pseudo-randomness design principle requires verifying that different models conforming to the specification are produced re-iterating the generation process multiple times with a limit on the number of expected results. The initial model consists of two *Class* instances. One class extends the other, which in turn contains an *Attribute* instance. Figure 21 illustrates the initial model.

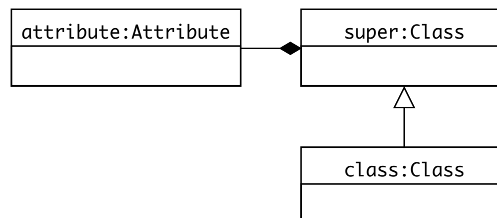

Figure 21: Pseudo-Randomness – Initial Model

The benchmark specification is illustrated in Listing 14 and included two optional differences, i.e. *Push Down Attribute* and *Rename Class*. No conflict exists among the differences, but only one model is expected to be generated.

```
benchmark
  .model("./resources/input/Input.xmi")
  .difference(0, 2, renameClass)
  .difference(0, 1, pushDownAttribute)
.build.generate(1, "./resources/output")
```

Listing 14: Pseudo-Randomness – Benchmark Specification

The benchmark generation process has been repeated 100 times, each iteration produced a difference combination among those illustrated in Figure 5.3 requiring an average time of 1,1 seconds to complete.

Visibility

Evaluating the visibility design principle requires verifying that all applied differences are clearly identifiable in the generated models, hence the framework discards results containing conflicting

and overlapping differences. The initial model consists of a *Package* instance containing a *Class* instance, as illustrated in Figure 22.

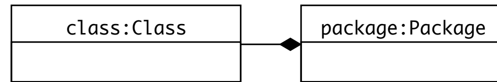

Figure 22: Visibility – Initial Model

The benchmark specification includes two mandatory and conflicting differences, i.e. consisting of contradicting modifications on the same model elements, as illustrated in Listing 15.

```
benchmark
  .model("./resources/input/Input.xmi")
  .difference(1, 1, deleteClass)
  .difference(1, 1, renameClass)
.build.generate(ALL, "./resources/output")
```

Listing 15: Visibility – Conflicting Differences

Intuitively, *deleteClass* overrides *renameClass*, i.e. obliterates all applied modifications if executed on the same model element, hence models containing both differences should not be generated. Given the initial model in Figure 22, executing the benchmark specification produced no solution, as expected.

Minimality

Evaluating the framework against the minimality design principle requires verifying that the minimal sequence of difference applications is preferred over the others in case of multiple paths producing the same model. The initial model consists of a *Package* instance containing a *Class* instance, as illustrated in Figure 23.

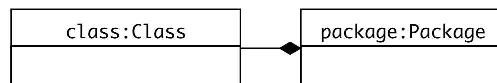

Figure 23: Minimality – Initial Model

The benchmark specification is illustrated in Listing 16 and includes three optional differences, i.e. *createClass*, *deleteClass* and *createAndDeleteClass*, which aggregates the previous ones.

```
benchmark
  .model("./resources/input/Input.xmi")
  .difference(0, 1, createClass)
  .difference(0, 1, deleteClass)
  .difference(0, 1, createAndDeleteClass)
.build.generate(ALL, "./resources/output")
```

Listing 16: Minimality – Benchmark Specification

The evaluation focused on the model resulting from the application of either the *createAndDeleteClass* or both the *createClass* and *deleteClass* differences. Observing the explored trajectories throughout the generation process, the first is correctly preferred over the latter given its shorter length. In case of bounded specifications, the property still holds as shorter trajectories are evaluated before longer ones adopting a breadth-first search strategy. Consequently, the difference trajectory consisting of both *createClass* and *deleteClass* will always produce a duplicated model and be discarded, i.e. already explored applying *createAndDeleteClass*.

6 Discussion

The main objective of this thesis consisted in providing support for the systematic evaluation of model comparison algorithms to researchers and practitioners. To this extent, a framework for the automated synthesis of model comparison benchmarks is proposed. Addressing the main research goal required providing an answer to the research questions presented in Section 1.1. The main contribution presented in this thesis can be mapped with the research questions as follows:

RQ1 – What kind of information is required for the specification of differences between models?

Understanding what kind of information is required for the specification of differences between models required observing the existing model comparison approaches with specific focus on the information and representation formalisms used to describe their results. The obtained findings served as basis for the development of a domain-specific language for the specification of model differences in terms of preconditions, actions and postconditions. In particular, actions describe the actual modifications that are applied, whereas preconditions and postconditions provide contextual information about the properties characterising the involved model elements.

RQ2 – How can we formulate the creation of model comparison benchmarks as design-space exploration problem?

Formulating the creation of model comparison benchmarks as design-space exploration problem required observing the existing contributions providing benchmarks for model comparison algorithms with specific focus on the recurring activities and involved artefacts. The obtained findings served as basis for the definition of essential design principles and properties required for model comparison benchmark generators. These principles have been used to identify the most suitable approach among the existing patterns used to automate model management tasks using design-space exploration techniques and to evaluate the existing contributions in model generation. Finally, a domain-specific language for the specification of model comparison benchmarks has been designed and a dedicated procedure mapping benchmark specifications to design-space exploration problem instances has been developed.

RQ3 – How can we represent the applied differences supporting their adaptation according to the requirements of an arbitrary comparison algorithm in analysis without loss of information?

Representing the applied differences to enable the evaluation of model comparison algorithms using different notations required a review of the existing contributions in model difference representation techniques. More specifically, the main objective consisted in identifying a difference representation formalism providing a low-level of abstraction to ensure no information would be excluded and a model-based notation enabling further manipulations using model transformations. Once identified the most suitable approach, a mapping procedure has been designed to construct difference representations from the solutions obtained by solving the design-space exploration problem.

7 Related Works

In this section, the existing contributions that somehow relate to the main contributions of this thesis are reviewed. First, the discussion focuses on approaches proposing manually built model comparison benchmarks or design guidelines to perform such task. The focus is then shifted towards the existing contributions presenting model generation languages or frameworks in the context of model-driven engineering.

Model Comparison Benchmarks

The existing contributions addressing the evaluation of model comparison algorithms generally propose manually constructed benchmarks focusing on specific modelling languages, differences and application domains.

In [18], the authors propose a benchmark for the evaluation of model comparison techniques over metamodels. More specifically, the benchmark is composed by a set of metamodels, each paired with other by a difference specification. Given its manual construction, the benchmark would be difficult to maintain or evolve in case the metamodels would change or other difference specifications would be required. Furthermore, the definition criteria and the granularity of the specified differences is not documented, hence the benchmark construction process per se is neither reproducible nor reliable. Finally, the benchmark only represent a useful source of information for the specific use case of metamodel comparison.

The model comparison benchmark proposed in [19] includes specific edit operations that have been observed to cause issues in existing model comparison algorithms. These operations are defined over fixed target modelling languages, hence narrowing the applicability of the benchmark. No difference model is provided to illustrate the correspondences among models, users would therefore need to manually reconstruct the benchmark, potentially introducing errors and biases.

The *M2BenchMatch* tool assists users in selecting the best performing comparison approach over a given pair of models [30]. Given the models and the expected differences among these, the tool executes and evaluates multiple comparison algorithms. However, requiring the user to provide the expected differences a priori seems to limit the practical usefulness of the tool. Despite automating the comparison of multiple algorithms, the user still needs to manually specify both the involved models and the correspondences among these. In this context, the framework proposed in this thesis might represent a convenient solution to generate input data for the tool.

Model Generation Frameworks

Various approaches addressing the automated synthesis of models have been proposed in the literature. In general, such approaches provide support for the specification of complex structural constraints that the generated models are expected to satisfy. The generated models are obtained starting from an initial model, creating instances conforming to a given metamodel or a combination of these approaches. The following paragraphs describe representative model generation approaches, Table 4 compares these against the design principles driving the development of the framework proposed in this thesis.

The *Epsilon Model Generation* (EMG) framework supports the specification of semi-automated model generators [44]. A specification is composed of creation and linking rules, where the first support the creation of model elements and the latter allow users to aggregate elements and specify complex structural constraints over these. Both number of expected models and element instances can be specified. However, the framework does not support the specification of an initial model. Furthermore, the generation process does not provide information regarding the operations that produced a given model, i.e. difference model. Finally, the framework does not provide support to guarantee that conflicting operations are avoided, hence that all specified operations are visible in the generated models.

Wodel is a domain-specific language for model mutation, where mutations consist in modified versions of an initial model [45]. Programs contains information regarding the number of expected mutants, the output folder where to store the generated models and the location of the initial

models. The actual modifications to perform are expressed in terms of mutation operators, which consist in sequences of built-in edit primitives. However, the language does not generate difference descriptions along with each mutation. Despite being possible to integrate post-processor components elaborating the generated models, constructing difference models a posteriori would require the assistance of a model comparison algorithm. Each mutant would need to be compared with the initial model, hence introducing clear dependencies on the chosen algorithm. Furthermore, the language does not ensure each applied modifications to be visible in the generated models, i.e. does not consider possible overlapping changes among conflicting operations, neither discards possible duplicated models from the resulting set.

The *SiDiff Model Generator* (SMG) is designed to support the controlled generation of realistic test models for model comparison tools [46]. Given a set of edit operations and an initial model, the tool selects and applies modifications using a probabilistic distribution resembling real-world edit interactions for a given application domain or modelling language. If needed, the generator is able to generate model histories, i.e. sets of consecutively modified models, and difference representations describing the applied changes. However, the framework does not guarantee the applied changes to be visible in the generated models and does not handle possible conflicts among these. Despite being possible to constrain the generated models, the notation used to represent model elements does not support reasoning about their changes over time.

The authors in [47] proposed a template language supporting the generation of test models with specific focus on testing the performance of model transformations. Template specifications describe the expected elements and structural properties of the generated models. The specifications are transformed and provided to an external model generator. Once produced, the generated models are transformed back to the initial formalism. This approach focuses on generating large-scale models to evaluate the scalability of model transformations. Models are generated from the ground up and specified in terms of the elements these should contain, rather than obtained applying user-defined differences over initial models. Consequently, the approach is not usable to evaluate model comparison tools. Intuitively, no difference description is generated as none is actually applied.

The model generator presented in [48] aims to provide support for evaluating the scalability of model management tools. Given a metamodel, the approach first constructs an equivalent tree representation. Using a random tree generation method on the representation, models with fixed size and uniform element distribution are then synthesized. Similarly to the previous one, the approach is not usable to construct model comparison benchmarks as it does not support the generation of models in terms of edit operations applied over initial models.

	EMG	Wodel	SMG	He et al. [47]	Mougenot et al. [48]	Benji
Configurability	✓	✓	✓	✗	✗	✓
Completeness	✓	✓	✓	✗	✗	✓
Pseudo-Randomness	✓	✓	✓	✗	✓	✓
Minimality	✗	✗	✗	✗	✗	✓
Visibility	✗	✗	✗	✗	✗	✓
Adaptability	✗	✗	✗	✗	✗	✓

Table 4: Model Generation Frameworks – Design Principles Comparison

8 Conclusion

The ever increasing adoption of model-driven engineering practices for complex software system development introduced the need for appropriate model management and evolution technologies. Model comparison, i.e. the identification of the differences existing among models, represents an essential task within numerous model management activities. The intrinsic hardness of detecting and analysing correspondences among models led researchers to propose a multitude of approaches, each proposing different approximation strategies exploiting structural, language-specific or domain-specific knowledge of the involved models. However, the large number of model comparison approaches corresponds to the almost complete absence of support for their systematic evaluation.

This thesis presents *Benji*, a framework for the automated synthesis of model comparison benchmarks. Given a set of difference specifications and an input model, users can generate mutant models resulting from the application of the first on the latter. Model differences are expressed in terms of preconditions, actions and postconditions. The generation process is designed to fulfil essential properties for model comparison benchmark generators and relies on design-space exploration techniques to produce the final solutions. Model differences are specified in terms of preconditions, actions and postconditions using a dedicated domain-specific language. Each of the generated models is associated with a model-based executable description of the applied changes.

Various benchmark use cases have been developed to evaluate the framework presented in this thesis. In particular, the evaluation aimed to stress its expressive support for the specification of both model comparison benchmarks and differences, while confirming its conformance to the essential properties for systematic model comparison benchmark generators. The evaluation proved the framework to fulfil all its design principles, unlike the existing model generation approaches.

Future Works

The languages composing the framework are currently implemented as embedded domain-specific languages. Despite representing a convenient design decision in terms of required implementation effort, the languages currently require users to define differences according to the search representation used throughout the generation process and might result verbose. Furthermore, the difference specification language currently require users to repeat preserved constraints in both preconditions and postconditions. In this context, possible future works might consist in implementing the framework over external domain-specific languages providing appropriate primitives for both actions, e.g. *create*, *delete*, *insert*, *remove*, and conditions, i.e. *created*, *deleted*, *inserted*, *removed*. Investigating the automated generation of metamodel-specific primitives also represent an interesting possible future work.

Generating model histories, i.e. sequences of consequent model versions, is currently not supported by the framework, which represent model elements in terms of initial and current versions only. Relaxing this constraint represents an interesting future direction of research, despite requiring non-trivial representation challenges to solve.

The framework has been evaluated focusing on its compliance with the essential properties of model comparison benchmark generators. However, further industrial experiments might provide essential feedback regarding the quality of the framework, e.g. scalability. In this context, valuable performance gains could result from parallelising the exploration process and merit further investigations. Moreover, the actual adoption of the framework for comparing existing model comparison algorithms might prove and provide further insights concerning the adaptability of the generated difference descriptions.

The ability to rapidly generate a large number of models, along with a description of their differences, which reflect characteristics of specific application domains, modelling languages and industrial policies paves the way for their usage in contexts different from that of evaluating existing model comparison algorithms. As future research direction, the generated artefacts might represent the training sets for model comparison algorithms integrating machine learning mechanisms to optimise their performance.

References

[1] J. Sztipanovits, S. Neema, and M. J. Emerson, "Metamodeling languages and metaprogrammable tools," in *Handbook of Real-Time and Embedded Systems*, 2007.

[2] J. Ludewig, "Models in software engineering - an introduction," *Software and Systems Modeling*, vol. 2, pp. 5–14, 2003.

[3] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, pp. 171–188, 2005.

[4] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, pp. 19–25, 2003.

[5] M. Alanen and I. Porres, "Difference and union of models," in *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, P. Stevens, J. Whittle, and G. Booch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–17.

[6] K. Altmanninger, M. Seidl, and M. Wimmer, "A survey on model versioning approaches," *IJWIS*, vol. 5, pp. 271–304, 2009.

[7] Y. Lin, J. T. Gray, and F. Jouault, "DSMDiff: A differentiation tool for domain-specific models," 2007.

[8] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[9] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Model comparison: a foundation for model composition and model transformation testing," in *GaMMa '06*, 2006.

[10] P. Kaufmann, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "An introduction to model versioning," in *SFM*, 2012.

[11] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pp. 1–6, 2009.

[12] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *Journal of Graph Theory*, vol. 1, pp. 339–363, 1977.

[13] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," 2008.

[14] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," *29th International Conference on Software Engineering (ICSE'07)*, pp. 54–64, 2007.

[15] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[16] L. Addazi, A. Cicchetti, J. D. Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Semantic-based model matching with EMFCompare," in *ME@MODELS*, 2016.

[17] M. Stephan and J. R. Cordy, "A survey of model comparison approaches and applications," in *MODELSWARD*, 2013.

[18] M. Wimmer and P. Langer, "A benchmark for model matching systems: The heterogeneous metamodel case," *Softwaretechnik-Trends*, vol. 33, 2013.

[19] P. Pietsch, K. Mueller, and B. Rumpe, "Model matching challenge: Benchmarks for Ecore and BPMN diagrams," *Softwaretechnik-Trends*, vol. 33, 2013.

[20] K. Vanherpen, J. Denil, P. D. Meulenaere, and H. Vangheluwe, "Design-space exploration in MDE: An initial pattern catalogue," in *CMSEBA@MoDELS*, 2014.

[21] Eclipse Modeling Framework (EMF) - Website. Accessed: 2019-05-22.

[22] Á. Hegedüs, Á. Horváth, I. Ráth, and G. Varró, "A model-driven framework for guided design space exploration," in *ASE*, 2011.

[23] L. Bettini, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Edelta: An approach for defining and applying reusable metamodel refactorings," in *MODELS*, 2017.

[24] M. Shaw, "What makes good research in software engineering?" *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 1–7, 2002.

[25] S. Tayal, "Engineering design process," *International Journal of Computer Science and Communication Engineering*, pp. 1–5, 2013.

[26] P. Pietsch, U. Kelter, and J. O. Ringert, "International workshop on comparison and versioning of software models (cvsm 2013) call for benchmarks," 2013.

[27] P. Pietsch, H. S. Yazdi, U. Kelter, and T. Kehrer, "Assessing the quality of model differencing engines," *Softwaretechnik-Trends*, vol. 32, 2012.

[28] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *EASE*, 2014.

[29] A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "A metamodel independent approach to difference representation," *Journal of Object Technology*, vol. 6, pp. 165–185, 2007.

[30] L. Lafi, J. Feki, and S. Hammoudi, "M2benchmatch: An assisting tool for metamodel matching," *2013 International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 448–453, 2013.

[31] C. Debreceni, I. Ráth, G. Varró, X. D. Carlos, X. Mendialdua, and S. Trujillo, "Automated model merge by design space exploration," in *FASE*, 2016.

[32] E. Kang, "Design space exploration for security," *2016 IEEE Cybersecurity Development (SecDev)*, pp. 30–36, 2016.

[33] Y. Xie, G. H. Loh, B. Black, and K. Bernstein, "Design space exploration for 3d architectures," *JETC*, vol. 2, pp. 65–103, 2006.

[34] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," *2009 Design, Automation and Test in Europe Conference and Exhibition*, pp. 423–428, 2009.

[35] T. Basten, E. van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. de Smet, L. J. Somers, and E. Teeselink, "Model-driven design-space exploration for embedded systems: The octopus toolset," in *ISoLA*, 2010.

[36] K. Römer and F. Mattern, "The design space of wireless sensor networks," *IEEE Wireless Communications*, vol. 11, pp. 54–61, 2004.

[37] K. Vanherpen, J. Denil, P. D. Meulenaere, and H. Vangheluwe, "Design-space exploration in model driven engineering : an initial pattern catalogue," in *MODELS 2014*, 2014.

[38] Á. Horváth and G. Varró, "Dynamic constraint satisfaction problems over models," *Software and Systems Modeling*, vol. 11, pp. 385–408, 2010.

[39] Xtend - Website. Accessed: 2019-05-22.

[40] G. Bergmann, Z. Ujhelyi, I. Ráth, and G. Varró, "A graph query language for emf models," in *ICMT*, 2011.

[41] S. Sen, B. Baudry, and J.-M. Mottu, "On combining multi-formalism knowledge to select models for model transformation testing," *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 328–337, 2008.

[42] Á. Hegedüs, Á. Horváth, and G. Varró, "A model-driven framework for guided design space exploration," *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 173–182, 2011.

[43] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: Elements of reusable object-oriented software," 1994.

[44] S. Popoola, D. S. Kolovos, and H. Hoyos, "Emg: A domain-specific transformation language for synthetic model generation," in *ICMT*, 2016.

[45] P. Gómez-Abajo, E. Guerra, and J. de Lara, "Wodel: a domain-specific language for model mutation," in *SAC*, 2016.

[46] P. Pietsch, H. S. Yazdi, and U. Kelter, "Generating realistic test models for model processing tools," *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 620–623, 2011.

[47] X. He, T. Zhang, M. Pan, Z. Ma, and C.-J. Hu, "Template-based model generation," *Software & Systems Modeling*, pp. 1–42, 2017.

[48] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria, "Uniform random generation of huge metamodel instances," in *ECMDA-FA*, 2009.