Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Systematic Evaluation of Model Comparison Algorithms using Model Generation

Lorenzo Addazi[a] Antonio Cicchetti[a]

a. School of IDT, Mälardalen University, Västerås, Sweden

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 analyse quality attributes and generate executable code combining the information from these. Raising the abstraction level to models requires 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. In this respect, almost no support is provided for the systematic evaluation of comparison approaches against specific scenarios and modelling practices, namely *benchmarks*.

In this article we propose *Benji*, a framework for the automated generation of model comparison benchmarks. In particular, by giving a set of difference patterns and an initial model, users can generate model manipulation scenarios resulting from the application of the patterns on the model. The generation support provided by the framework obeys specific design principles that are considered as essential properties for the systematic evaluation of model comparison solutions, and are inherited from the general principles coming from evidence-based software engineering. The framework is validated through representative scenarios of model comparison benchmark generations.

Keywords Model Comparison Benchmark; Model Comparison; Model Matching; Model Differencing; Model Generation; Design-Space Exploration; Model-Driven Engineering;

1 Introduction

Model-Driven Engineering (MDE) promotes the migration from code-centric to model-based software development. Traditional software engineering practices envision models

Lorenzo Addazi, Antonio Cicchetti. *Systematic Evaluation of Model Comparison Algorithms using Model Generation*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 11:1–22. doi:10.5381/jot.2020.19.2.a11

as mere documentation artefacts facilitating the communication among stakeholders. Model-driven approaches, instead, represent systems as collections of interconnected models focusing on specific concerns [28]. Models assume the role of first-class citizens throughout the development process, quality attributes can be evaluated earlier and executable code is (semi-)automatically generated from these [11].

Shifting from source code to models has led intense research concerning model management and evolution techniques in the recent years [35]. In this context, model comparison represents one of the most challenging tasks and is essential to various other model management activities, e.g. model versioning [5, 6], model transformation testing [24], and model synchronisation [27].

Model comparison consists of two main phases, i.e. matching and differencing [21]. Most of the complexity commonly attributed to model comparison results from its matching phase. Indeed, matching elements from different models can be reduced to finding an isomorphism among their corresponding typed graph representations, which is known to be an NP-Hard problem [33]. Although numerous approaches have been proposed in the literature, these inevitably represent different attempts to provide an approximate solution by exploiting structural [22, 12], language-specific [44], or domain-specific [27, 4] knowledge about the involved models [6, 37]. In other words,

> *"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. [25]

In this context, researchers and practitioners require support for evaluating and comparing different approaches against specific application domains, languages and modelling practices. Unfortunately, the large number of model comparison techniques available in the literature corresponds to the almost complete absence of support for their systematic evaluation. Here, for systematic evaluation we refer to a process supported by the fundamental principles entailed by evidence-based software engineering [23] and empirical studies in it [42]. In this respect, it shall be possible to properly define comparison scenarios, called *benchmarks* in this article, against which selected quality attributes of the available solutions would be evaluated.

Manually defined benchmarks, e.g. [41] and [29], do not provide a plausible solution for every use case as modelling language, application domain, applied differences and their representation are fixed. Furthermore, their design rationale is often not accessible and potentially biased, thus making the results unreliable and not reproducible. On the other hand, even when existing models for a benchmark were available for reuse, it would still be required to both precisely identify the expected comparison results and encode those results into appropriate representations to enable the measurements for a specific comparison approach. Therefore, the assessment of model comparison algorithms becomes a time-consuming and error-prone task requiring deep knowledge about both modelling language, application domain, and comparison algorithm under evaluation. In a broader perspective, manual practices limit the possibilities of adopting systematic approaches for the evaluation of model comparison solutions [23, 42].

In this article, we first identify the essential properties for model generation approaches supporting the systematic generation of domain-specific model comparison benchmarks. Based on these, a framework for the automated synthesis of domain-specific model comparison benchmarks is designed and developed. Given a set of difference specifications and an input model, design-space exploration is used to

produce model mutants resulting from the application of the former ones on the latter. For each mutant, a description of the applied changes is also generated. Differences are represented in terms of preconditions, actions and postconditions. Preconditions and postconditions assert properties of the involved model elements before and after their modification, respectively. Actions, instead, provide an operational specification of the edit statement to execute. Eventually, various benchmark construction examples integrating metamodel refactoring patterns [10] have been designed and implemented to evaluate our automated synthesis framework.

The remainder of this article is structured as follows. Section 2 introduces model comparison and its current evaluation practices. Section 3 illustrates the identified properties for model generators in the context of model comparison benchmark synthesis. Our proposed framework is then presented providing details as concerns expected input, execution and produced results. In section 4, the framework is evaluated against various examples re-adapting well-known metamodel refactorings on a simplified version of the Ecore metamodel [38]. Section 5 compares our contribution with related works either addressing the assessment of model comparison algorithms or generating models with user-defined properties. Section 6 concludes the paper summarising the obtained results and drawing possible future work directions.

2 Model Comparison

Model comparison consists in detecting differences between the models given as input. The existing approaches can be distinguished with respect to the number of models involved in the comparison process, i.e. two-way and n-way techniques. The first class groups approaches limiting their comparison to two models, i.e. initial and current, whereas the latter supports an arbitrary number of models. The remainder of this work focuses on two-way model comparison.

Given two models, model comparison algorithms produce a difference representation illustrating the changes among these. As depicted in Figure 1, the process is decomposed into two phases – matching and differencing [21]. First, all elements from the first are linked with the corresponding ones in the latter. Then, a difference representation is constructed processing the identified correspondences[1].

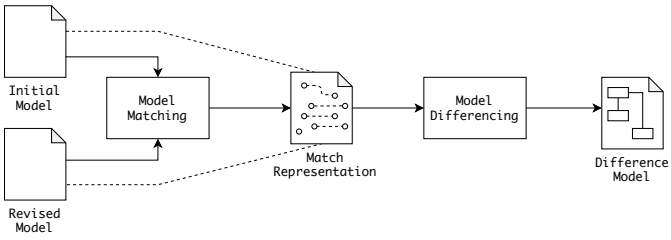

Figure 1 – Model Comparison

[1]It is worth noting that some works distinguish between representation and visualisation, where the latter is the rendering of the detected evolution for the user in an appropriate format [12]. However, since the benchmark synthesis does not impact visualisation per se, in the rest of this work we keep representation and visualisation merged as defined in [21].

2.1 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 the matching criteria, the existing approaches are grouped in four categories – identity-based, signature-based, similarity-based, and language-specific [25]. Identity-based and signature-based techniques assign persistent or dynamically generated unique identifiers to model elements, respectively. The matching criteria consists in mapping elements with the same identifier and the similarity score is binary. Similarity-based algorithms, instead, compare model elements using similarity functions defined over their structural features. Unlike the previous approaches, mappings are associated with continuous values (i.e. likelihood scores), hence many-to-many matches are possible. Eventually, language-specific approaches extend similarity-based matching by exploiting semantic information of specific application domains or modelling languages to optimise their matching process or refine the obtained results.

2.2 Differencing

Model differencing consists in translating the identified mappings into meaningful change descriptions. The existing techniques are grouped in two categories – operational and denotational differencing. The first represents changes in terms of edit primitives applied on the initial model, whereas the latter (also known as state-based) describes changes in terms of visible effects in the final model. *Edit Scripts* represent a well-known metamodel-independent operational approach involving primitives to create, delete and update elements – namely new, delete, set, insert, remove [5]. The notation presented in [13], 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 X, three corresponding metaclasses are generated representing created, deleted and changed instances, respectively – *ChangedX*, *CreatedX* and *DeletedX*.

2.3 Evaluation of comparison quality

The concept of quality in model comparison algorithms is relative to the model management activity these are integrated into, and the evaluation criteria used to assess their results depend on the specific use case [31]. For example, low-level difference representations might result convenient in semi-automated workflows, e.g. model transformation testing. However, the same difference would result hardly readable and inconvenient in tasks involving human reasoning, e.g. manual conflict resolution in model versioning.

Regardless of the evaluation criteria, 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 positives and true positives – as illustrated in Table 1. Negatives are differences that have not been identified during the comparison

process, whereas positives consist in differences that have been identified. Both partitions are further decomposed into 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, namely precision, recall, and f-measure (Equation 1–3). In particular, precision and recall compute the percentage of correct differences over all those detected, and of correctly detected differences over all the expected ones, respectively. Besides, 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 over precision and recall. Notably, the *overall* metric shown in Equation 4 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 [26]. Intuitively, this metric aims to measure the effectiveness of a given model comparison algorithm within a model versioning workflow.

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

In order to enable the systematic evaluation of comparison algorithms against quality attributes like the metrics mentioned so far, it is necessary to adopt rigorous experimental set-ups as described in empirical software engineering state-of-the-art [42]. In particular:

- benchmarks shall be clearly linked to a purpose, i.e. the properties to measure and the comparison scenarios in which they shall be evaluated;
- benchmarks shall be randomised, e.g. to avoid biases;
- benchmarks shall enable validity evaluations, that is how much the measurements can be generalised to specific comparison problems.

According to these principles, we propose a synthesis framework for the automated generation of comparison benchmarks. The generation process is driven by user-defined descriptions of comparison scenarios and delivers as output random modifications conforming to those scenarios.

3 The proposed benchmark synthesis framework

In this section, we first introduce a set of design principles for model generators to support the automated synthesis of domain-specific model comparison benchmarks. We identify these principles as essential to enable the systematic evaluation of comparison algorithms. Based on the principles, the discussion proceeds illustrating *Benji*, a framework for the systematic generation of model comparison benchmarks: first, the overall framework architecture is presented; then, further details concerning runtime model representation, involved domain-specific languages, and synthesis process are provided.

3.1 Design Principles

The essential design principles concerning model comparison benchmark generators involve both the specification formalism provided to the user, the generation process, and the representation of the final results. In particular, they originate from the state-of-the-art definition of *experiment process* in empirical software engineering [42] by considering the *definition*, *planning*, and *operation* phases, i.e. those impacted by the automated generation of comparison benchmarks. Subsequently, these have been refined and specialised by reviewing the current practices in creating benchmarks for model comparison algorithms combined with recurring features and limitations characterising the existing model generation and model comparison approaches. In this context, it is important to notice that the principles concern the model generation framework and not the possible benchmarks constructed by integrating the generated models with some metrics of interest for the user.

Configurable – Model comparison benchmarks are constructed aggregating triples, each consisting of an initial model, a modified version and a description of the differences among these. Model generation approaches should support the specification of initial model and expected differences. Furthermore, users should be able to configure minimum and maximum number of expected applications for each difference, the number of models to generate, and the location where to store them.

Complete – Given a benchmark specification, the generation process should perform an exhaustive exploration of the solution space. More specifically, model generation approaches should consider all possible combinations of differences within their respective minimum and maximum application bounds. Considering the definition provided in [40], a generation process is complete if any finite instance model of a domain can be generated in finite steps. In our case, the domain is composed of the initial model and the set of available differences, i.e. the possible model manipulations and their lower/upper bounds.

Pseudo-Random – Model generation approaches should prevent users from introducing biases during the synthesis process. At any given time, a pseudo-random selection criteria should be adopted in selecting the difference to apply. Users should

not be able to influence the selection process in case the same difference was applicable multiple times.

Minimal – Duplicated models do not provide additional value in model comparison benchmarks, hence model generators should ensure no duplicates are generated. In case the same modified model might be obtained through multiple sequences of difference applications, the minimum length trajectory (i.e. the smallest set of modifications) should be selected.

Visible – As a direct consequence of state-based versioning, difference applications might overwrite or hide the results of previous ones involving common model elements. Model generation approaches should handle overlapping and conflicting differences throughout the synthesis process ensuring their visibility in the final models.

Adaptable – Model comparison algorithms adopt different notations and granularity in representing their results. Model comparison benchmark generators should produce low-level and model-based change descriptions, hence allowing their conversion to algorithm-specific notations using semi-automated transformations and without requiring external information to be integrated.

Table 2 summarises the relationships between the design principles discussed so far and the phases in experiment processes mentioned above and defined in [42]. It is worth noting that *analysis* and *presentation* phases are not included since they are related to activities following the generation of comparison benchmarks, namely the selection of metrics and the corresponding measurement of comparison algorithms quality attributes.

	Definition	Planning	Operation
Configurable	✓	✓	✓
Complete	✗	✗	✓
Pseudo-Random	✗	✗	✓
Minimal	✗	✗	✓
Visible	✗	✗	✓
Adaptable	✓	✓	✓

Table 2 – Relationships between design principles and experiment process

3.2 Overall Architecture

Model comparison benchmarks are specified in terms of initial model and expected differences using a dedicated domain-specific language, as illustrated in Figure 2. Given a benchmark specification, the synthesis process consists of two phases – (A) difference-space exploration and (B) output construction.

During the difference-space exploration phase, the benchmark specification is used to build and solve an equivalent design-space exploration problem instance. The obtained solutions are transformed into pairs of output models and difference representations. Aggregating these pairs with the initial model produces a model comparison benchmark conforming with the initial specification.

The framework is implemented using the Eclipse Modelling Framework (EMF) [1], and Viatra-DSE, a design-space exploration framework for EMF models [17]. However, the fundamental concepts and mechanisms proposed in this article abstract from implementation details and specific technological choices.

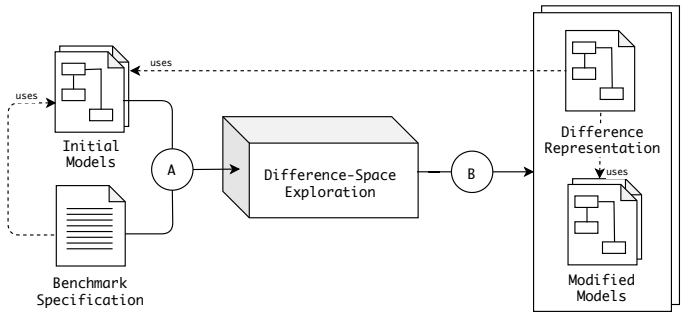

Figure 2 – Framework - Overview

3.3 Trace Representation

Representing changing model elements throughout the synthesis process requires an appropriate formalism providing read-write access to their current state while maintaining a read-only representation of the initial state. In our framework, model elements are wrapped into trace objects and their representation is split into initial and current version, as illustrated in Figure 3.

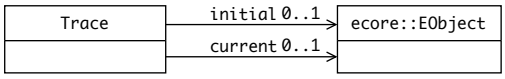

Figure 3 – Traces - Metamodel

Model elements can be identified into three possible states – created, deleted and preserved. Intuitively, created and deleted elements are represented as trace objects without initial and current object links, respectively. Preserved elements, instead, are represented as trace instances where the links target two different instances representing initial and current state of the model element taken into account. 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 4 provides an example of a renamed model element, i.e. where the name attribute value has been changed.

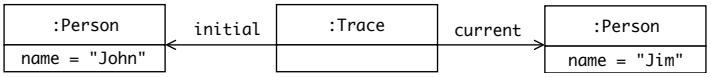

Figure 4 – Traces – Changed Element

3.4 Benchmark Specification

Benchmarks and differences are specified using dedicated domain-specific languages built on top of existing technologies and designed to fulfil the *Configurable* and *Visible* properties discussed in Section 3.1.

3.4.1 Difference Specification Language

Ensuring difference visibility requires the framework to provide means to express properties concerning the involved model elements before and after its application, in

addition to the actual modifications to perform. The visibility of a given difference can then be verified ensuring that the corresponding postcondition is satisfied in the current model.

Figure 5 illustrates the abstract syntax of the difference specification language in our framework. Model differences consist of actions, preconditions and postconditions. An action consists of imperative edit statements to execute whenever the corresponding difference is applied, whereas precondition and postcondition describe properties of the involved model elements before and after the difference is applied, respectively.

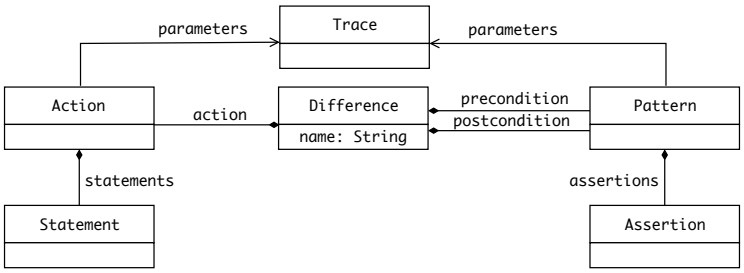

Figure 5 – Difference Specification Language – Metamodel

The difference specification language is implemented as embedded domain-specific language in Xtend, a flexible and expressive dialect of Java [2]. Preconditions and postconditions are expressed using VQL, a domain-specific language for the specification of patterns over EMF models [9].

Starting from a *Person* metaclass with a *name* string attribute, a possible renaming difference specification is described in the following paragraphs. Each difference specification starts with a unique name attribute, whereas precondition, action, and postcondition are specified referencing the corresponding external definitions. In this example, the action updates the current attribute value prepending a constant string (i.e. `changed`) to the initial one, as shown in the right side of Figure 6.

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

(a) Difference

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

(b) Action

Figure 6 – Rename Precondition - Difference and Action

The difference shall involve a single person instance such that its name attribute has not been changed before. This precondition is specified as shown in Figure 7 left-hand side. First, initial and current instance of the preserved person are retrieved (2 – 3). Then, their corresponding name attribute value is accessed (4 – 5). Finally, the precondition verifies that initial and current name attribute value are equivalent (6). Once applied the action, the involved person instance shall be preserved, while its current name shall differ from the initial one. Figure 7 right-hand side illustrates the corresponding postcondition, of which the only difference with respect to the precondition is that initial and current name attribute values shall now be different (6).

```
1 pattern beforeRenamePerson(person) {
2  Trace.initial (person, init_person);
3  Trace.current (person, curr_person);
4  Person.name (init_person, init_name);
5  Person.name (curr_person, curr_name);
6  init_name == curr_name;
7 }
```

(a) Precondition

```
1 pattern afterRenamePerson(person) {
2  Trace.initial (person, init_person);
3  Trace.current (person, curr_person);
4  Person.name (init_person, init_name);
5  Person.name (curr_person, curr_name);
6  init_name != curr_name;
7 }
```

(b) Postcondition

Figure 7 – Rename Person - Precondition and Postcondition

3.4.2 Benchmark Specification Language

The model comparison benchmark generator should be configurable, hence support the specification of initial models and differences with minimum and maximum number of expected applications. Furthermore, users should be able to specify an upper bound on the number of generated models and the location where to store them.

Figure 8 illustrates the abstract syntax of the benchmark specification language included in our framework. Benchmark specifications consist of an initial model and a set of bounded differences. The initial model is referenced using its resource location, whereas bounded differences correlate existing difference specifications with a non-negative integer lower and upper bound on the number of expected applications. Benchmark instances also keep track of maximum number of models to synthesize and output path in the filesystem.

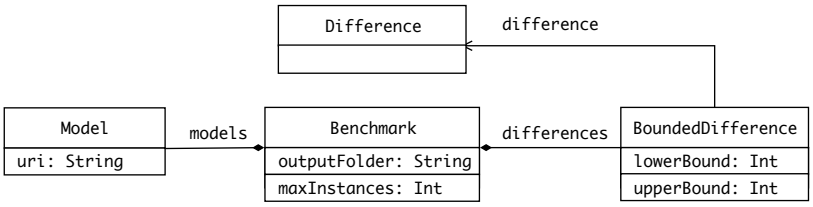

Figure 8 – Benchmark Specification Language - Metamodel

The benchmark specification language is implemented as embedded domain-specific language in Xtend. Figure 9 provides a sample benchmark specification expecting three kinds of differences to be applied, 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, respectively. Each difference is associated with minimum and maximum number of expected applications, e.g. *renamePerson* is optional and can be applied at most two times 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.

3.5 Difference-Space Exploration

Difference-space exploration represents the core stage of the generation process. Given a benchmark specification, this task consists in searching for those modified versions of the initial model containing the expected differences. In our framework, this phase

```
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")
```

Figure 9 – Benchmark Specification Language - Example

is realised constructing and solving a design-space exploration problem instance. The choice of formulating the process as a design-space exploration problem is based upon the successful application of these techniques to automate similar tasks requiring the search for models with specific characteristics, e.g. model transformation testing [36].

In our framework, model comparison benchmark specifications are mapped to rule-based design-space exploration problem instances [18]. In these, each problem consists 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, solutions consist in sequences of model manipulations, i.e. *solution trajectories*. The following paragraphs illustrate how benchmark specifications are mapped into constraint-satisfaction problems over models.

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. Initial and current version of a given model element are completely equal at the beginning of the exploration process.

Goals – Lower and upper bounds associated with each difference 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 occurrences of difference D within a 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.

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. Moreover, 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 the upper bounds. Consequently, 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 this respect, the framework shall also check and ensure its satisfaction throughout the exploration process. Global constraints are verified over all intermediate and final solutions, hence represent a suitable mechanism. Given a solution trajectory T, the following global constraint verifies that the visibility 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 $POST(D)$ checks that difference D is visible in the current model.

Model Manipulations – Model manipulations rules are automatically extracted from the differences listed in a given benchmark specification. In detail, preconditions are mapped to guards and actions as rule bodies.

3.6 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 model 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 10.

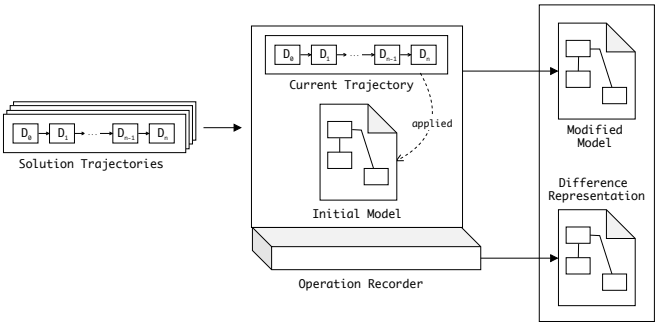

Figure 10 – Framework - Output Construction

The operation recorder is currently implemented using *EMF.Edit*, the standard EMF change monitoring support utilities [1]. The applied differences are represented using the internal *EMF.Edit* mechanism, that adopts a state-based approach similar to the one proposed in [13] (see Section 2.2). In this way, by considering the design principles discussed in Section 3.1, the framework fulfils the adaptable property by providing low-level and model-based difference representations.

4 Evaluation

In order to evaluate the framework proposed in this article against the design principles presented in Section 3.1, various model comparison benchmark generation use cases have been designed. In particular, the evaluation process consisted in implementing the refactoring patterns composing the *Metamodel Refactoring Catalogue* using *Simplified Ecore* as base metamodel [38]. We have chosen this catalogue due to several aspects: it includes both "primitive" and "complex" manipulations, thus

stimulating the expressiveness of the benchmark definition languages, their usability, and for validating principles like completeness and visibility; it targets a very well investigated evolution problem for MDE that still poses challenges and is tightly related to model manipulation detection. For each design principle, corresponding benchmark specifications have then been defined, executed and verified.

4.1 Simplified Ecore

In *Simplified Ecore*, models consist of root packages containing all other elements. Packages are uniquely identified by their *Universal Resource Identifier* (URI) attribute value and might contain three possible types of instances: packages, classes and datatypes. Classes represent the core modelling concept of the metamodel. Each instance contains attributes and references. An opposite reference can be defined in the latter case. Finally, all elements provide a name attribute extending the NamedElement metaclass. The metamodel is illustrated in Figure 11.

```
abstract class NamedElement {
  String name
}

class DataType extends NamedElement {}

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 References opposite
}
```

Figure 11 – The Simplified Ecore Metamodel

4.2 Metamodel Refactorings Catalogue

In the metamodel refactoring catalogue, operations are classified with respect to two aspects – granularity and operation types. 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 involve multiple operations. Table 3 illustrates the complete refactoring catalogue along with information regarding whether or not the refactoring has been successfully implemented in the evaluation.

All refactorings have been successfully implemented using our framework. Among the implemented patterns, those in the table that are marked with a star (*) required complex reasoning in their precondition and postcondition. The increasing complexity 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 [8].

4.3 Design Principles

In the following, we recall the essential properties defined in Section 3.1 and discuss exemplary benchmark specifications to illustrate how the evaluation of our framework has been performed against those properties[2].

[2]The interested reader can access the full replication package at `https://github.com/loradd/se.mdh.idt.benji.examples`

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 Catalogue

Complete

Evaluating the framework against the completeness design principle required verifying that models resulting from all bounded difference combinations are generated. The same results must be produced on multiple iterations if no limit on the number of models is provided. The initial model is illustrated in Figure 12 and consists of two class instances. One class extends the other, which in turn contains an attribute instance.

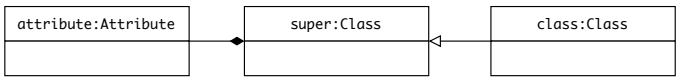

Figure 12 – Complete – Initial Model

The benchmark specification is illustrated in Figure 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")
```

Figure 13 – Complete– Benchmark Specification

The generation process terminated in 1,2 seconds and correctly produced all distinct models resulting from all non-empty difference combinations as illustrated in Figure 14. In particular, each item is a set of one or more differences, distinguished by their name and applied to a selected object in the initial model. So for example, the first set of generated differences is a *renameClass* metamodel refactoring applied

to the instance called *class*, while the last item generated is a set including three manipulations: *renameClass* metamodel refactorings applied to *class* and *super* and a *pushDownAttribute* refactoring involving all the items included in the input model.

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

Figure 14 – Complete – Expected Difference Combinations

Pseudo-Random

Evaluating the framework against the pseudo-random design principle required verifying that different models conforming to the specification are produced re-iterating the generation process with a limit on the number of expected results (without a limit the generation would produce each time the same entire set of possible modifications due to completeness).

The initial model is borrowed from the completeness validation as depicted in Figure 12. The benchmark specification is shown in Figure 15 and includes two optional differences, i.e *Push Down Attribute* and *Rename Class*. No conflict exists among the differences, and differently from the completeness validation sample 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")
```

Figure 15 – Pseudo-Random – Benchmark Specification

The generation process has been repeated 100 times, each iteration produced a difference combination among those listed in Figure 14 and required an average time of 1,1 seconds to complete.

Visible

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 16.

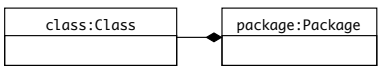

Figure 16 – Visible – Initial Model

The benchmark specification includes two mandatory and conflicting differences, i.e. consisting of contradicting modifications on the same model elements, as illustrated in Figure 17.

In this example, *deleteClass* overrides *renameClass*, i.e. obliterates all applied modifications if executed on the same model element, hence models containing both

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

Figure 17 – Visible – Benchmark Specification

differences should not be generated. Given the initial model, executing the benchmark specification produced no solution as expected.

Minimal

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.

For the initial model we consider again the one introduced in Figure 16. The benchmark specification is illustrated in Figure 18 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")
```

Figure 18 – Minimal – 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, since already explored by applying *createAndDeleteClass*.

5 Related Work

This section discusses existing literature related to the framework proposed in this work. In particular, the discussion focuses on approaches proposing manually built model comparison benchmarks or design guidelines to perform such task. Moreover, existing model generation languages or frameworks in the context of model-driven engineering are illustrated.

Model Comparison Benchmarks

The vast literature related to model comparison algorithms generally proposes manually constructed benchmarks focusing on specific modelling languages, differences, and application domains. In [41], 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, paired with each 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 is specifically meant for metamodel comparison use cases.

The model comparison benchmark proposed in [29] 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. Since no difference model is provided to illustrate the detailed correspondences among the involved models, users would 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 [26]. Given the models and the expected differences among them, the tool executes and evaluates multiple comparison algorithms. However, requiring the user to provide the expected differences a priori can limit the practical usefulness of the tool. In fact, despite automating the comparison of multiple algorithms, the user still needs to manually specify both the involved models and the correspondences among them, a tedious and error-prone task especially when needing large experiments. In this context, the framework proposed in this work might represent a convenient solution to generate input data for the *M2BenchMatch* 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, or by creating instances conforming to a given metamodel, or through combinations of both of them. The following paragraphs describe representative model generation approaches, while Table 4 compares them against the design principles driving the development of the framework proposed in this work.

The *Epsilon Model Generation* (EMG) framework supports the specification of semi-automated model generators [32]. 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 them. 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 models. 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 of modified versions of an initial model [15]. Programs contain 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, that are 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. In fact, each mutant would need to be compared with the initial model, hence introducing undesired influences due to the chosen algorithm. Furthermore, the language does not ensure each applied modification to be visible in the generated models, i.e. it 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 [30]. 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 them. In fact, 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 [16] 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.

	EMG	Wodel	SMG	[16]	Benji
Configurable	✓	✓	✓	✗	✓
Complete	✓	✓	✓	✗	✓
Pseudo-Random	✓	✓	✓	✗	✓
Minimal	✗	✗	✗	✗	✓
Visible	✗	✗	✗	✗	✓
Adaptable	✗	✗	✗	✗	✓

Table 4 – Model Generation Frameworks – Design Principles Comparison

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 DSE techniques have been proposed and successfully applied in different domains, e.g. model merging [14], software security [20], circuit design [43, 19], embedded system development [7, 34]. In practice, three main classes of approaches integrating MDE with DSE techniques have been identified – model generation, model adaptation and model transformation [39]. 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. The DSE mechanism exploited by the framework proposed in this paper pertains to the model

adaptation class[3].

6 Conclusion and future works

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 relying on 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 article proposes the automated synthesis of model comparison benchmarks as a way to systematically evaluate model comparison approaches, and hence to support a well-founded experimentation process [23, 42]. In this respect, it first introduces a set of principles for the design of appropriate model generation mechanisms, and then it presents a framework meeting those principles, called *Benji*. In particular, 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 by using a dedicated domain-specific language. Moreover, the generation process relies on design-space exploration techniques to produce the final solutions. Each of the generated models is associated with a model-based executable description of the applied changes. The approach has been validated through various benchmark use cases, aimed to both stress its expressive support for the specification of model comparison benchmarks and differences, and to confirm its adherence to the properties for systematic model comparison benchmark generators.

Possible future works include the implementation of external domain-specific languages to make the specification of pre-/postconditions and model manipulations more user friendly. Moreover, it could be interesting to generate model histories, i.e. sequences of consequent model versions, even though we expect to face non-trivial representation challenges to solve. Eventually, 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 robustness of the framework, e.g. scalability, usability. Performance gains might result from introducing task-level parallelism in the exploration process and merit further investigation. The achieved speedup is inversely proportional to the frequency of synchronisation among tasks, hence the effectiveness of introducing parallelism here depends on the extent to which the exploration process can be restructured to minimise inter-task dependencies. Finally, 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.

[3]For the sake of space, we refer the reader to [3] for a thorough discussion about why this class suits better than the others our benchmark synthesis purposes.

References

[1] Eclipse Modeling Framework (EMF) - Website, 2020. Accessed: 2020-02-09.

[2] Xtend - Website, 2020. Accessed: 2020-02-09.

[3] L. Addazi. Automated synthesis of model comparison benchmarks, 2019.

[4] 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.

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

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

[7] 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.

[8] G. Bergmann. Translating ocl to graph patterns. In *MoDELS*, 2014.

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

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

[11] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

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

[13] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6:165–185, 2007.

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

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

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

[17] 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)*, pages 173–182, 2011.

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

[19] 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*, pages 423–428, 2009.

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

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

[22] 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*, 97:1–14, 2014.

[23] B. A. Kitchenham, T. Dybå, and M. Jorgensen. Evidence-based software engineering. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 273–281, USA, 2004. IEEE Computer Society.

[24] 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.

[25] 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*, pages 1–6, 2009.

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

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

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

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

[30] 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)*, pages 620–623, 2011.

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

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

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

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

[35] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20:19–25, 2003.

[36] 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*, pages 328–337, 2008.

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

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

[39] 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.

[40] D. Varró, O. Semeráth, G. Szárnyas, and Á. Horváth. *Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models*, pages 285–312. Springer International Publishing, Cham, 2018.

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

[42] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

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

[44] 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.

About the authors

Lorenzo Addazi is a Ph.D. student in parallel modelling and programming languages for heterogeneous embedded systems at IDT, Mälardalen University (Sweden). His research interests also include model-driven software engineering, model versioning and comparison, hybrid modelling, domain-specific languages, software language engineering and programming models. Contact him at `lorenzo.addazi@mdh.se`, or visit `http://www.es.mdh.se/staff/3276-Lorenzo_Addazi`.

Antonio Cicchetti is an Associate Professor at the Industrial Software Engineering research group in IDT, Mälardalen University, Västerås (Sweden). His current research topics include domain-specific and general-purpose modelling languages, model transformations, and the management of consistency in multi-view/multi-paradigm design frameworks. Moreover, he works on the introduction/enhancement of model-based techniques in industrial contexts. Contact him at `antonio.cicchetti@mdh.se`, or visit `http://www.es.mdh.se/staff/198-Antonio_Cicchetti`.