JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Domain-Specific Model Distance Measures

Eugene Syriani[a] Robert Bill[b] Manuel Wimmer[c]

a. Université de Montréal, Canada
b. CDP, TU Wien, Austria
c. CDL-MINT, Johannes Kelper University Linz, Austria

Abstract Much research was invested in the last decade to develop differencing methods to identify the changes performed between two model versions. Typically, these changes are captured in an explicit difference model. However, quantifying the distance between model versions received less attention. While different versions of a model may have the same amount of changes, their distance to the base model may be drastically different. Therefore, we present distance metrics for models. We provide a method to generate tool support for computing domain-specific distance measures automatically. We show the benefits of distance measures over model differences in the use case of searching for the explanation of model evolution in terms of domain-specific change operations. The results of our experiments show that using distance metrics outperforms the usage of common difference models.

Keywords Model comparison, Model diffing, Model distances, Model evolution

1 Introduction

The emergence of model-driven engineering (MDE) [Sch06, BCW17] has increased the need for dedicated techniques for model management [KRM$^+$13]. In particular, much research was invested in the last decade to develop differencing methods to identify the changes performed between two model versions. As surveyed in [SC13], most algorithms aim at computing differences and representing them in the form of difference models which capture the changes between model versions. Difference models are critical in MDE, being used for various model management tasks, such as metamodel/model co-evolution, versioning or synchronization [DRIP12, DRELHE16, TELW14].

While most work has been focusing on differences, quantifying the distance between model versions received less attention. Distances are useful in addition to differences for several reasons. First, while different versions of a model may have the same amount of differences, their distance to the base model may be drastically different. Second, distances can be an additional metric to reason about the evolution paths of models to reach a specific setting. As an example, consider the movement of attributes between different classes in a class diagram. For example, suppose we have classes A, B, C, and D connected in sequence with associations. If we move an attribute from A to any other class, we always get the same difference: the attribute deleted from A and added to one of the other classes. However, a distance metric could tell us *"how far"* we have moved the attribute away from A, leading to different distance measures depending to which class it has moved.

In this work, we present the notion of distance metrics for models as an additional measurement of the difference between models. Furthermore, we provide a method to derive distance metrics tailored to the domain-specific language (DSL) at hand. We implemented a software library to automatically generate domain-specific model distance calculators, given the metamodel and the change operators of the DSL. We apply the distance metrics on the use case of searching for the explanation of model evolution in terms of domain-specific change operations. Our results show that using distance metrics outperforms the usage of common difference models, although domain-specific distance measures have their own challenges.

In Section 2, we overview the background of our approach and motivate our work with a running example for our use case. In Section 3 we present how to compute the model distance metrics and how to derive them for a particular DSL. In Section 4, we briefly outline our implementation and the use case for the following evaluation section. In Section 5, we evaluate the application of these metrics on our use case. We discuss related work in Section 6 and conclude in Section 7.

2 Background and Motivation

Like any software artifact, models evolve continuously. Knowing the operations applied between two successive versions of a model is crucial for helping developers to efficiently understand the evolution [KHL$^+$10]. It is also a major prerequisite for model management tasks [KDRPP09]. In general, we distinguish between two categories of model differencing approaches to report model differences. One describes model differences as a set of generic individual operations, while the other uses domain-specific operations that aggregate individual ones to be applied as a single operation.

2.1 Model differences as atomic operations

Current model comparison tools often apply a two-phase process to compare a base model and a revised model. First, model matching algorithms compute the correspondences between elements of the two models to compare [KDRPP09]. Then the model differencing phase computes the differences between two models from the established correspondences. For instance, EMF Compare [BP08]—a prominent representative of model comparison tools in the Eclipse ecosystem—can detect the following types of atomic operations:

- Add: A model element only exists in the revised version.
- Delete: A model element only exists in the base version.
- Update: A feature of a model element (attribute value or reference) has a different value in the revised version than in the base version.
- Move: A model element has a different container in the revised version than in the origin version.

The advantage of differencing atomic operations is the generic tool support it provides to work for any models of any DSL. Its limitation is when there is a large number of atomic differences, which may require a higher level of abstraction. Therefore, the following differencing approaches have been developed over the last years.

2.2 Model differences as domain-specific operations

To raise the level of abstraction of model differences, we can aggregate the atomic operations into domain-specific operation applications, making the intent of the change explicit [SPTJ01]. Existing solutions, e.g., see [LWB$^+$13, KKT11, XS06, VWV11] to mention just a few, provide language-specific operation detection algorithms. Often, executable domain-specific operations are specified as model transformations. Transformation rules define the preconditions, postconditions, and actions of the operation. Especially, the approaches proposed in [LWB$^+$13, KKT11] build on model transformations to detect the operations performed to obtain the revised model from the base model. The output of these approaches is a sequence of transformation rule applications corresponding to the domain-specific operation.

In these approaches, the difference model can be compressed into showing fewer differences than with atomic differences [LWB$^+$13]. However, finding the set of domain-specific operations which best describes a model evolution is a challenging problem, since there are many different evolution paths between two versions and there are dependencies between the execution of the operations which may mask some of them in the revised version of a model. Therefore, search-based approaches have been developed to evaluate different evolution paths to find the revised model [bFKLW12, KMW$^+$17] without having to search for operation occurrences in atomic diff models. Nevertheless, these approaches rely on atomic differences to compare the computed model versions with the given revised model in the fitness function used by the search algorithms.

2.3 Motivating and Running example

We rely on the running example of a simplified Pacman game, a well-known game where Pacman navigates through grid nodes searching for food to eat, while ghosts try to kill him. We implemented a DSL to define game configurations, based on [SV13]. Figure 1 shows the metamodel and Figure 3 illustrates three Pacman game models in the concrete syntax of the DSL. Pacman, food, and ghosts are placed on grid nodes with an `on` reference. Grid nodes are connected by `left`, `right`, `up`, and `down` references to define the permissible navigation of Pacman and ghosts. The concrete syntax represents references by topological alignment rather than arrows. In M1, Pacman is on grid node 21 which has an `up` reference to grid node 11 and a `right` reference to grid node 22. A score object keeps track of the points of every food Pacman eats. Red and green food score for 1 and 3 points respectively. We define the operational semantics of the DSL in terms of an inplace model transformation, implemented with graph transformation rules as in [SV13]. One rule represents Pacman eating food on a grid node and updating the score. Another represents the ghost killing Pacman when they are on the same grid node. Four rules for Pacman and four others for the ghost represent moving in each direction to an

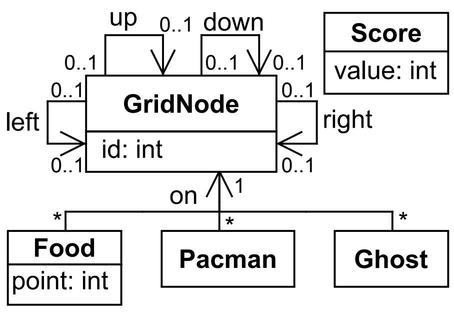


Figure 1 – The metamodel of the Pacman game DSL

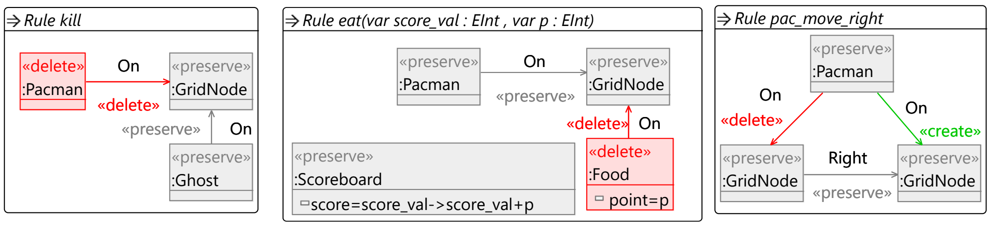


Figure 2 – The kill, eat, and move right rules of the Pacman game

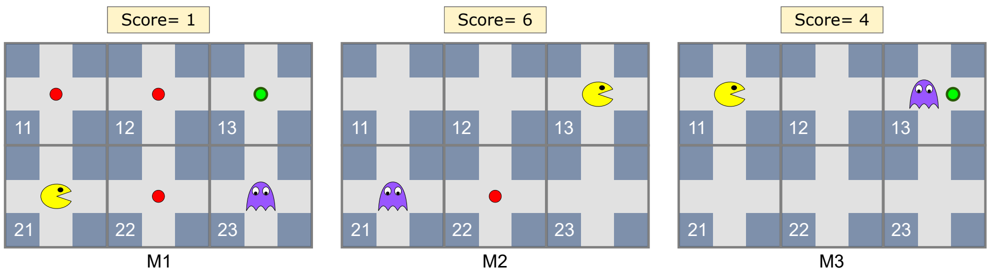


Figure 3 – The initial model M1 and two possible models resulting from applying different rules of the Pacman game

adjacent grid node. Figure 2 illustrates some of these rules in Henshin[1]. Although the rules should obey certain scheduling, e.g., killing has priority over moving to end the game, in this work, we assume that the transformation is a graph grammar, i.e., any rule may be applied at any time its precondition is satisfied during the execution of the transformation.

For our use case, we are interested in finding the minimal sequence of rule applications starting from the initial model leading to the target model. Search-based techniques are very useful to solve this problem. They explore large parts of the search-space by generating intermediate models as a result of applying the rules while optimizing the objective to get closer to the target model. For example, a minimal rule sequence, of length 8, to go from M1 to M2 in Figure 3 is Pacman moves up once, then moves right twice, eating the food each time, while the ghost also moves left twice. A minimal rule sequence, of length 7, from M1 to M3 is Pacman moves right, then up, then left, eating the food each time, while the ghost moves up once.

Detecting the rule sequence requires to compare models at every step. To compute the difference between M1 and M2, a generic model comparison tool, like EMF Compare, would report that three food objects are deleted, two `on` references (Pacman and ghost) are changed, and the attribute value of the score is modified. This tool would report the same aggregate information when we compare M1 and M3. However, a domain expert would immediately detect that there is a clear difference between the two situations: 8 rule applications are needed to obtain M2 and 7 to obtain M3. Furthermore, the score value hints to which type of food Pacman ate. Thus, the information output by common model differencing approaches is not precise enough to identify the minimal sequence of rules (e.g., for the M2 case, if Pacman moves right first, he will have to go through grid node 12 at least twice). The main reason is that they rely solely on changes in the abstract syntax. However, the comparison needs to be tailored to both the DSL and its semantics to find the best rule sequence. In this example, relying on creation, deletion, and modification of elements of the metamodel is not

[1] Henshin: `http://www.eclipse.org/henshin`

sophisticated enough. The notion of Pacman and ghost movements as well as a quantification of the score value must be encoded in the comparison. Inplace model transformation rules typically encode change operations, such as operational semantics or refactoring [LAD$^+$16]. Therefore, we propose a set of domain-specific distance metrics that take into consideration abstract syntax changes as well as the semantics of the transformation to provide more precise information when comparing models. It would allow optimizing the search space exploration of identifying the minimal sequence of rule applications.

3 Domain-specific Model Distances

3.1 Model distance metrics

Typical model difference tools report metrics on elements added and deleted in terms of instances of metamodel classes, on references changed in terms of instances of metamodel associations, and on attribute value modifications. As motivated in Section 2.3, we need metrics tailored to the DSL both in terms of the metamodel and the semantics of the transformation. Therefore, we propose the following three model distance metrics.

The semantics of many modeling formalisms relies on movements of model elements. Some of their elements are *movable* while others represent *positions* where an element can move to. Pacman moving on the grid in our example, attributes moving to superclasses in class diagrams, or tokens moving between places in a Petri net are some of many examples where it happens. Furthermore, some elements are *modifiable* meaning that the transformation changes some of their attribute values, like the score in the rule where Pacman eats food.

Formally, we represent a model as a labeled, attributed multi-graph $G = \langle V, E, l, a\rangle$. We identify three subsets of nodes $Mov, Pos, Mod \subseteq V$ corresponding to the movable, position, and modifiable objects in the model. In our running example, grid nodes are the position nodes while Pacman and ghosts are movable nodes. Note that, in general, Pos may be different for each $v \in Mov$. Additionally, all three subsets do not form a partitioning of E: an object can move between positions, but it can also serve as a position for other movable objects, and it can have attributes modified. Among the set of edges $e : V \to V \in E$, we identify two subsets $N, P \subseteq E$. Neighbor edges $n : Pos \to Pos \in N$ only connect position nodes, e.g., `left`, `right`, `up`, and `down` references between grid nodes. Position edges $p : Mov \to Pos \in P$, like the `on` references, connect a movable node to a position node. The label function $l : V \to \Sigma^*$ assigns a unique string label to each node, where Σ is a set of alphanumeric characters. l is used to identify corresponding elements in two model versions. The attribute function $a : V \times \Sigma^* \to Value$ assigns values to each attribute name of a node, such as the `value` of the score and the `point` of a food object. Listings 1 and 2 show how M1 and M2 are encoded in the graph structures G_1 and G_2 respectively.

Listing 1 The formal graph structure G_1 of model M1 in Figure 3

$Pos_1 = \{grid_{ij}\}_{i,j=1..3}, Mov_1 = \{pacman, ghost\}, Mod_1 = \{score\}$

$V_1 = Pos_1 \cup Mov_1 \cup Mod_1 \cup \{f_i\}_{i=1..4}$

$(grid_{11}, grid_{12}), (grid_{12}, grid_{11}), (grid_{12}, grid_{13}), (grid_{12}, grid_{11}), (grid_{11}, grid_{21}), \ldots \in N_1$

$(pacman, grid_{21}), (ghost, grid_{23}), (f_1, grid_{22}), (f_2, grid_{11}), (f_3, grid_{12}), (f_4, grid_{13}) \in P_1$

$E_1 = N_1 \cup P_1$

$a(score, \text{‘}value\text{’}) = 1, a(f_4, \text{‘}point\text{’}) = 3, a(f_i, \text{‘}point\text{’}) = 1, i = 1..3, \forall v \in V_1 : l(v) = \text{‘}v\text{’}$

Listing 2 The formal graph structure G_2 of model M2 in Figure 3

$Pos_2 = Pos_1, Mov_2 = Mov_1, Mod_2 = Mod_1, V_2 = Pos_2 \cup Mov_2 \cup Mod_2 \cup \{f_1\}$
$(pacman, grid_{13}), (ghost, grid_{21}), (f_1, grid_{22}) \in P_2, N_2 = N_1, E_2 = N_2 \cup P_2$
$a(score, \text{‘}value\text{’}) = 6, a(f_1, \text{‘}point\text{’}) = 1, \forall v \in V_2 : l(v) = \text{‘}v\text{’}$

3.1.1 Move distance

The move distance of a movable object is the length of the shortest path from its position in model M1 to its position in model M2. To compute the move distance between M1 and M2, we identify the common connected subgraph G_{12} of their respective graphs G_1 and G_2. We define $\delta_M(m_1, p_2)$ as the length of the shortest sequence of neighbor edges connecting $p(m_1)$, the position of the movable node $m_1 \in Mov_1$ to the position node $p_2 \in Pos_2$, where Mov_i, Pos_i correspond to subsets of nodes in a graph G_i. In the M2 case in Figure 3, $\delta_M(pacman, grid_{13}) = 3$ and $\delta_M(ghost, grid_{21}) = 2$, which is equivalent to the Manhattan distance in this grid layout. We define the move distance between two models as:

$$\Delta_M(G_1, G_2) = \sum_{m_1 \in Mov_1, m_2 \in Mov_2} \delta_M(m_1, p(m_2)), \text{ where } l(m_1) = l(m_2)$$

Here, $p(m_2)$ is the position of the movable object corresponding to m_1 in G_2. The move distance assumes that $p(m_1)$ and $p(m_2)$ are connected by a path of neighbor edges. If such a path does not exist, then $\delta_M(m_1, p(m_2)) = \infty$. In our example, $\Delta_M(M1, M2) = 5$ and $\Delta_M(M1, M3) = 2$. We rely on the popular Floyd-Warshall algorithm [Flo62, War62] to compute the shortest path between two nodes in a connected graph. The dynamic programming implementation takes $O(|Pos|^3)$ to compute all distances between any two nodes and $O(|N|)$ to output the path.

3.1.2 Element distance

This metric is concerned with the presence and absence of metamodel class instances between M1 and M2. It is similar to what a model difference algorithm outputs. We define the element distance as:

$$\Delta_E(G_1, G_2) = \frac{|\{v_1 \in V_1 | \nexists v_2 \in V_2, l(v_2) = l(v_1)\}| + |\{v_2 \in V_2 | \nexists v_1 \in V_1, l(v_1) = l(v_2)\}|}{|V_1| + |V_2|}$$

The numerator counts the number of nodes exclusively in each graph. To normalize the distance as a ratio between 0 and 1, we divide by the total number of nodes in both graphs. In our example, M1 is composed of a Pacman, a ghost, four food objects, six grid nodes and a score, thus its graph has 13 nodes (see Listing 1). Therefore, $\Delta_E(M1, M2) = \Delta_E(M1, M3) = \frac{3+0}{13+10} = 0.13$. We can interpret this distance as the ratio of objects added or removed between the two models. Note that the element distance is not concerned with edges since they are already considered by the move distance.

3.1.3 Value distance

The third metric is concerned with the difference in attribute values between objects in M1 and M2. We define the value distance of attribute x of node v between G_1 and G_2 as $\delta_V(v, x)$ by comparing $a(v, x)$ and $a(v_1, x)$, where $v_1 \in Mod_1, v \in Mod_2$ and $l(v) = l(v_1)$. δ_V returns a positive real number and is left to the user to define a custom distance function for specific attributes of the metamodel. By default, if the attribute value can be encoded as a unique number, we define the value distance as the margin of error needed to obtain the value of x in v from its value in v_1.

$$\delta_V(v,x) = \begin{cases} |a(v_1,x)| & \text{if } a(v,x)=0 \\ |a(v,x) - a(v_1,x)|/a(v,x) & \text{otherwise} \end{cases}$$

Here, we only consider attributes of objects present in both M1 and M2 because the element distance already takes care of the absence and presence of elements. Note that if $a(v,x) = 0$ we replace the denominator by 1. We define the value distance $\Delta_V(G1,G2)$ between two models as the average of δ_V for all attributes of all nodes in G_{12}. It calculates the average margin of error between all attribute values of the two models. In our example, $\Delta_V(M1,M2) = \delta_V(score, value) = \frac{|4-1|}{4} = 0.75$ and $\Delta_V(M1,M3) = 0.86$.

Typically, if a model difference tool reports the same changes in M2 and M3, the element distance will be the same for both cases as well. However, the move and value distance will typically discriminate the two as we have seen in the Pacman example. Furthermore, distance metrics provide a quantitative approximation of the difference in terms of *"how far"* (thus, comparison distance) M1 is from M2.

3.2 Adapting distance metrics to the DSL

The distance metrics presented in Section 3.1 are generic model distances to compare two models. We now describe how to adapt these metrics for a particular DSL and its semantics. We aim to produce a distance calculator given the metamodel of the DSL and a set of inplace model transformation rules encoding its semantics. Typically, these rules have a precondition and a postcondition pattern.

We need to identify the metamodel classes corresponding to the sets of nodes Pos and Mov, and the associations corresponding to the sets of edges N and P. The potential candidates for Mov are classes that have an association to another class in the metamodel with cardinality at most 1. We denote A the potentially movable class and r its association to the other class B. Instances of A, r, and B must be in the precondition of a rule and r must be modified in the postcondition to reference another class instance. Then, potentially A is a class of movable nodes, r is a position edge type, and B is a class of position nodes. In our example, these are, among others, the `Pacman` class, the `on` association, and the `GridNode` class, respectively. It is also possible that r is an association from B to A. Furthermore, it may be that the second instance A refers to is of another type than B, say C. If there is a reference s between B and C, then s is likely to be a neighbor edge. Note that this is a necessary condition but not sufficient. For example, it may be the case where the movable and position classes are connected directly but through an intermediate class.

Similarly, we analyze the classes of the metamodel such that the postcondition of a rule modifies one of its attributes value. Such classes define the type of the nodes in Mod.

The value distance metric is entirely configurable to fit the domain, especially if we know a priori the distribution of the values of an attribute, or if the distance of pairwise non-numeric values can be partially ordered (such as an enumeration of colors).

The three distance metrics rely on the label function l to correspond similar nodes between the two models. For example in Figure 3, the grid nodes are identified by their identifier (e.g., 11, 12, ...). However, not all classes in the metamodel of the DSL have an identifying attribute. Since the label function must uniquely identify each node, we must compute a label for each object that does not have one. We can compute the label structurally using different strategies [KKPS12]. For example, we can ascertain that there is at most one food on a grid node. Then the label of a food object can rely on the label of the grid node it is on. Another case is if we can ascertain that a class is a singleton, then we assign the same label to its instance.

The method we describe to characterize the sets and functions required for the distance metrics must be adapted to each DSL. Therefore, this method serves as recommending suggestions to the developer who needs to manually refine the characterization to be specific to the DSL at hand.

4 Use Case: Finding a Sequence of Rule Applications

As motivated in Section 2.3, we apply domain-specific distance metrics to the problem of finding the minimal sequence of rule application from an initial model M1 to a target model M2. We follow previous work [bFKLW12, KMW$^+$17] and consider this problem as an optimization problem and use search-based techniques to solve it.

Having the evolution recovery problem at hand, we apply our search-based framework MOMoT [FTW16] to find the Pareto-optimal model evolutions. MOMoT[2] is a task- and algorithm-agnostic approach that combines SBSE and MDE. It has been developed in previous work [FTW16] and builds upon Henshin [SBG$^+$17] to define model transformations and the MOEA framework[3] to provide optimization techniques. In MOMoT, DSLs (i.e., metamodels) are used to model the problem domain and create problem instances (i.e., models), while model transformations are used to manipulate those instances. The orchestration of those model transformations, i.e., the order in which the transformation rules are applied and how those rules need to be configured, is derived by using different heuristic search algorithms which are guided by the effect the transformations have on the given objectives. For instance, MOEA provides an implementation of NSGA II [DAPM02], a multi-objective genetic programming framework. In order to apply MOMoT for the given problem, we need to specify the necessary input: two model versions, change operators defined as Henshin rules, and the objectives for the search.

We use four objectives for the search for operation sequences encoded in the fitness function of the search-based algorithm. First, we want to minimize the length of the sequence of rules applied. Then, we minimize the three distance metrics move, element, and value tailored to the DSL. These substitute the common difference metrics used in [bFKLW12, KMW$^+$17].

From Section 3.2, we understand that automatically adapting the distance metrics to the DSL is very challenging. Therefore, we implemented the distance metrics as a Java library[4] `EModelDistance` that computes the metrics on any Ecore model in the Eclipse Modeling Framework. The library encapsulates all dependencies on the metamodel of the DSL within one abstract class `DistanceUtility`. The developer must override the five sets Pos, Mov, Mod, N, and P for her DSL. She must also define the `getId()` function for each class of the metamodel, if it does not already have an identity attribute. The computation of the metrics is generated into Java classes that rely on the customized `DistanceUtility` class provided for that DSL.

In our current implementation of the distance metrics, we do not explicitly merge the two models. Instead, we assign a distance of ∞ when there is no neighboring edge between a position node in G_1 and G_2 if G_{12} required it.

Next, we evaluate potential performance differences of using common difference metrics, e.g., provided by EMF Compare, and the distance metrics introduced in this paper.

[2] MOMoT: `http://martin-fleck.github.io/momot`
[3] MOEA Framework: `http://www.moeaframework.org`
[4] `https://github.com/geodes-sms/EModelDistance`

5 Evaluation and Discussion

We evaluate our approach by comparing how domain-specific distance metrics perform compared to difference models for model management tasks. In particular, we focus on the use case presented in Section 4, namely finding a sequence of change operation applications between two versions of a model. When searching for change operations, we apply changes to a base model and measure how close it is to the given revised model. Thus, the fitness function has to foresee a component for this decision. We perform different experiments to see if the usage of model distances is in favor of using difference models in the fitness function.

5.1 Objectives

The comparison, distance or difference, is encoded in the fitness function of the search for operation-based changes. Thus, the objective of this evaluation is to assess the impact of the fitness functions on the search process. In particular, we are interested in the following research questions:

- **RQ1—Search-space exploration:** Does a fitness function using distance metrics find better solutions than one using model differencing?
- **RQ2—Search time:** Do distance metrics help in finding the best solutions in fewer iterations than model differencing?

For RQ1, we analyze the final solutions output by the search-based algorithm for each model comparison approach. For RQ2, we analyze the intermediate results at each iteration of the search.

5.2 Experiment setup

To ensure a fair comparison of the experiments, we use the same base/revised models, transformations, solution length, and optimization algorithms for all experiments. They only differ in the fitness function used. We performed our experiments on three commonly used case studies. In each one, we generated the domain-specific metric computation from the `EModelDistance` from the library we implemented. We opted for EMF Compare to calculate the difference model. On top of that, we built a basic distance function calculating the similarity based on the fraction of matched elements and attributes divided by the number of all elements and attributes. We define the metamodels in Ecore and the semantic change operations as model transformation rules in Henshin. For each case, we randomly generated multiple instances of the metamodel, corresponding to the base and revised models. Models, rules, and metamodels are all input in MoMOT, as described in Section 4, to find the minimal sequence of operations that led to the revised model version.

5.2.1 Cases.

The following are the three cases we considered for our experiments.

Pacman. We presented the Pacman case in Section 2.3. The test models we considered consist of 8×8 grid nodes with one Pacman, three ghosts, and 15 food objects all randomly distributed on the grid.

Table 1 – Characterization of the three cases

	Pacman	**Petrinet**	**Refactoring**
Rule count	High	Low	Medium
Rule complexity	Low	Low	High
Solution length	High	High	Low
Position changes	None	N/A	High

Petri nets. A Petri net model consists of places connected to other places via transitions. Places can hold tokens that can be transferred to other places by firing transitions. A place can have multiple transitions. Each transition can have multiple outgoing places, contributing to non-deterministic transition firing. We model places and transitions as objects and tokens as integer attributes of places.

Here, we only use one transformation rule that allows moving a single token from a place to another place. Although this is an oversimplification of the Petri nets formalism, this rule will be applied many times, especially when there are tokens in multiple places. The test models we considered consist of 12 places and, on average, two transitions per place, and one to two outgoing places per transition. The net is initialized with one to two tokens per place distributed randomly across places.

Object-oriented refactoring. For this case, we have reused a variant of the solution[5] to the 2013 edition of the Transformation Tool Contest (TTC). Here, the metamodel defines classes which can have named, typed attributes and generalizations. The type of an attribute can be primitive or an existing class. This example has been modified to make stronger use of the domain-specific distance metrics presented in this paper. For the experiments, we focus on a subset of the refactoring rules. *Pull up attribute* moves an attribute from all subclasses into their superclass. *Extract superclass* creates a new intermediate superclass if an attribute exists in some, but not all, subclasses of a particular superclass. *Create root class* is similar to the latter if the classes do not already have a superclass.

Please note that the terminology used in the TTC 2013 edition is deviating from the common object-oriented refactoring terminology which uses the term "extract superclass" for the refactoring behaviour provided by the "create root class" refactoring of the TTC 2013 edition. In order to be able to reuse the refactoring rules as they are, we further use the terminology as introduced by the TTC 2013 edition in the remainder of this paper.

We added a new rule *Move attribute* that moves an attribute from one class to another class it is associated with. The test models we considered consist of nine classes, one to three attributes per class. Each attribute has a name chosen from a set of seven possibilities and a type from three possibilities. Each class has a 50% chance to be generalized with a superclass.

We selected these three case studies because they differ in terms of the number of rules, the complexity of each rule, and the expected solution length (see Table 1). The Petri net case contains only a single rule which can match in many different ways and its application does not prevent its re-execution. Thus, the rule count is low, but the expected solution length is high. In contrast, the rules for the refactoring case typically can be applied in a more limited way, yielding a lower expected solution length. Nevertheless, this case heavily changes the structure of the model by adding classes and generalizations, as well as removing attributes. The Pacman case is in between: although it has more rules, they semantically move around Pacman and ghosts, but differ in what happens on specific grid nodes.

The Petri net case does not create or remove any objects at all. The Pacman case removes

[5] Refactoring: `http://martin-fleck.github.io/momot/casestudy/class_restructuring/`

objects like food or Pacman itself. However, it does not change the connections of the position and neighbor edges. The refactoring case creates classes which are position objects for attributes; thus it modifies the structure on which the attributes move. The last row of Table 1 reflects this difference.

5.2.2 Methodology

When experimenting with search-based algorithms, we typically execute the same algorithm on the same model many times and compare averages. However, in our experiments, we only want to compare the values of the fitness functions. Therefore, we opted to count the number of transformation rules applied when the revised model is found; otherwise we set it to infinity. Thus we cannot compare averages, but pair-wise fitness function values for the domain-specific distance and the difference model approach. Nevertheless, we use 100 base and expected revised model pairs for each approach on each case study.

We generated the base models randomly within the parameters described in each case. To generate the revised model, we applied the transformation rules randomly, making sure they are meaningful and usable (e.g., Pacman is not killed right away).

Two runs of two identical randomized algorithms might produce results of the same or different quality. If the results are not equal, the probability that either the first or the second run is better is 50%. Thus, the probability that k first runs were better than $n-k$ second runs can be calculated using the binomial distribution as $\binom{n}{k}0.5^k 0.5^{n-k} = 0.5^n\binom{n}{k}$. The probability that k or fewer runs were better can be calculated using the binomial distribution function with $p = 0.5$, or rather $F(k\|p,n) = \sum_{i=0}^{k}\binom{n}{i}0.5^i 0.5^{n-i} = 0.5^n\sum_{i=0}^{k}\binom{n}{i}$.

Our null hypothesis is that the fitness function using difference models finds a better sequence of operations leading to the revised model than if it were using domain-specific distance metrics. The binomial distribution gives us the probability that the former would only be better in k or fewer cases by chance. We can reject this null hypothesis if we get better results in at most 5% of all cases.

5.3 Results

In the following, we describe the results of the experiments conducted in terms of the research questions.

5.3.1 RQ1—Search-space exploration

Table 2 compares the quality of the results obtained by each approach out of the 100 runs. The first three columns denote the number of times the domain-specific distance metrics yielded a better result, a result taking the same amount of transformations, or a worse result than the generic difference model, based on the values of the fitness functions. The fourth column denotes how many times both fitness functions failed to find the revised model. All p-values are under 0.05; therefore we can reject the null hypothesis.

The search had a hard time finding the best solution for the Pacman case for both approaches, which explains the high number of "no exact solution". However, out of the remaining 28 runs, the distance fitness function was able to find better solutions almost all the time. We observed a similar situation for the refactoring case. After manual inspection, we found only 15 cases where both algorithms found a valid solution of the same quality. Still, the domain-specific distance was better in 80% of all cases. For the Petri net case, the domain-specific distance approach found a good result most of the time. The difference model approach could not find any result in 57% of the runs, whereas the distance approach failed only 26% of the runs.

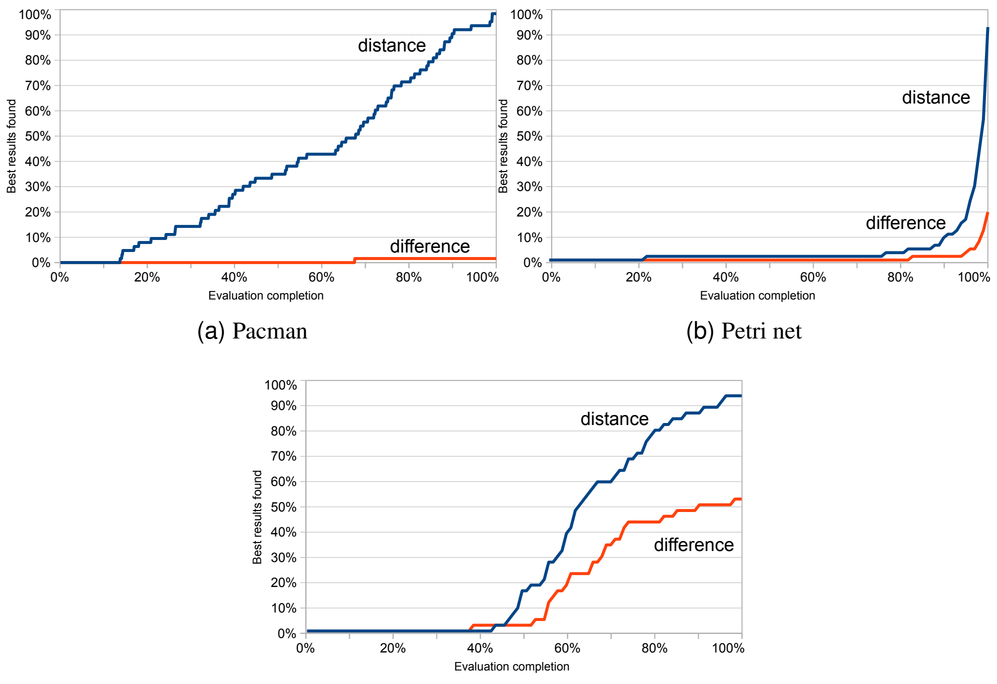
Table 2 – The results of the experiments for RQ1

	Better	**As good**	**Worse**	**No exact solution**	***p*-value**
Pacman	27	0	1	72	$1.1 \cdot 10^{-6}$
Petrinet	62	2	12	24	$1.4 \cdot 10^{-9}$
Refactoring	32	15	8	45	$9.1 \cdot 10^{-5}$

We can therefore positively answer RQ1: domain-specific distance metrics yield better results. Yet, it is not so clear when the models change their position and neighbor edges. Nevertheless, it performs best when movable objects change or attribute values change.

5.3.2 RQ2—Search time

To answer RQ2, we only consider runs that find the expected revised model correctly. Fig. 4 shows the evolution of the fitness functions in terms of the iterations of the search algorithm for each case. The x-axis represents the search progress in terms of the percentage of completed iterations, i.e., the current iteration count divided by the total number of iterations needed to find the best solution for each of the 100 runs. The y-axis shows the percentage of runs for each fitness function which have already found the best model for the run, independent of the fitness function. Each data point represents the average result from all 100 runs. If one fitness function outperforms the other, i.e., yields a better or equal value for each run, it would show 100% on the y-axis at the right side of the graph. If the other fitness function also manages to be better in some scenarios, it will be lower than that.



(a) Pacman

(b) Petri net

(c) Refactoring

Figure 4 – Stepwise comparison of the different experiments

The results of the Pacman case show a clear advantage for the domain-specific distance metrics. On the one hand, they help Pacman to eat the correct amount of food. On the other hand, they allow both Pacman and the ghosts to have a clear direction in reaching their target position. Thus, this result concurs with the result for RQ1. For the Petri net case, the results are continually improving by finding better solutions each time. The graph appears to hint that the search process only finds the right model at the end. However, a manual inspection reveals that the search finds the right model early, but it finds better sequences of rules just before the search ends. In the Refactoring case, we can see that it takes half the amount of iterations to start finding models it cannot improve later on for both approaches. The reason is that the position objects (classes) must first be created and deleted, where the distance metrics are the same for the EMF compare and the domain specific distance function, before attributes can start moving correctly, where the domain specific distance function actually helps. As the first step already takes about half the iterations, there is little difference between both fitness functions in that time. Afterwards, the domain-specific distance metrics continuously outperforms the difference model. In total, the former finds 85% of the best results while the latter only finds 50% of them.

We can therefore positively answer RQ2: domain-specific distance metrics improve the speed of the search process. The difference is more apparent for examples which can make better use of the move and value distances. That is because the element distance is very similar to the metrics based on EMF Compare. In particular, the Pacman case performs best as all of its rules directly correspond to changes in the domain-specific distance. The other cases show only weaker use of the domain-specific distance and thus less clear result. The Petri net case uses attribute values, but as these values are seldom large, there is less difference between a coarse-grained generic attribute change and a more fine-grained value change distance metric. The Refactoring case uses transformations that do not necessarily improve the domain-specific distance.

While we did not measure the exact execution times, our approach was always faster than EMF Compare. While EMF Compare was configured to match on identifiers, the generic comparison function still had to evaluate the complete model instead of focusing on the dynamic parts. While the graph-based distance function evaluation could take longer for larger models, it could be precomputed for static structures, which are the kinds of models for which the domain-specific distance function works best.

5.4 Threats to validity

As we use randomized algorithms which may produce different results on each run, we use 100 runs for each model. The structure of our test models allows us to use a simple statistical test without any requirements for metrics like the variance of the fitness values. While we achieved statistical significance with a one-sided test allowing 5% error, which may be considered as one of the weakest acceptable values, the p-values report the same significance trend with a two-sided test with 1% error.

Differences in the search process which are not due to the difference in the distance metrics used could cause a problem for the internal validity of our experiments. However, we have ensured that as many parameters of our search are constant by using the same algorithm, the same input models, the same search framework and the same search parameters. While the number of objectives is different, we chose NSGA II as our search algorithm as it works both for two and four objectives.

Currently, the approach seems to work best when the transformation mainly changes either attribute values or moves objects around. Several preliminary tests with the original refactoring case have suggested that there might be little or no advantage when the operations mainly modify position nodes and neighbor edges. This is probably due to both distance and difference approaches individual elements similarly. Therefore, we needed to modify the case (as explained in Section 5.2) to allow both fitness functions to find better solutions.

Another threat is that we only compare one representative of the domain-specific distance approach (EModelDistance) to only one representative of the difference model approach (EMF Compare). Nevertheless, since EMF Compare is commonly used and our implementation is the only one currently available, we can state that the results we obtain from each implementation are representative of each approach.

5.5 Discussion

We can conclude that the domain-specific distance can help the search process in various cases. It is especially useful if the transformations exclusively move objects, but can also provide benefits if some transformations perform other changes.

The current implementation assumes that the base and revised models have the same position objects (none are deleted or created). However, as seen with the Refactoring case, this is too restrictive. The models should be preprocessed by merging their position elements and then use the move distance. One possibility is to use EMF Compare to merge them based on the position elements.

While we can recover a plausible sequence of change operations for arbitrary base and revised models, this trace is not necessarily the one originally used to produce the revised model. For example, if Pacman and the ghosts are on opposite sides of the board and move together so that Pacman is killed, they can meet at any position in between and have the same result model and the same length of the sequence of rule applications as we don't require that the game stops when Pacman is killed.

6 Related Work

We first discuss related work on model differencing and then on approaches that cluster model based on distance metrics when analyzing model repositories.

6.1 Model Differencing

Kolovos et al. [KDRPP09] survey different model matching approaches. Model matching is often the first phase of model differencing processes. Model matching can be: *static identity-based*, which assume a unique identifier for objects; *signature-based*, which compare objects based on a dynamic signature calculated from the objects' properties; *similarity-based*, which match objects based on the aggregated weighted similarity of their properties, but obviates the model semantics; and *language-specific*, developed for a modeling language and its semantics. For example, using *signifiers* [LWG$^+$12] (i.e., combinations of features of a metamodel class) as comparison criteria falls under the signature-based category. EMF Compare is similarity-based but permits defining custom matching algorithms. In our work, we did not tackle model matching and reused existing techniques for this purpose. However, as an extension, one could use model distances for the matching process too.

Many approaches attempt to derive model differences formulated as atomic operations. For example, [SC13] surveys several model comparison approaches and [TELW14] proposes a formalization. In [CRP07], the authors derive a DSL from metamodels to capture differences based on atomic changes. This is in line with our approach, but we provide dedicated support to compute distance measures. To the best of our knowledge, the existing approaches that compute atomic changes do not compute model distances.

There is also a dedicated line of research concerned with the detection of domain-specific operations. For instance, Xing and Stroulia [XS06] present an approach to detect refactoring operations in evolving software models. They integrated their work with UMLDiff. The approach by Vermolen et al. [VWV11] copes with the detection of complex evolution steps between different versions of a metamodel. They use a difference model comprising primitive changes as input to calculate complex changes. They even detect changes that are hidden by other changes such that their effect is partially or totally missing in the revised model. Gerth et al. [GKLE13] calculate hierarchical change logs including compound changes in the absence of recorded changelogs. They apply the concept of Single-Entry-Single-Exit fragments to calculate the hierarchical changelogs after computing the correspondences between two process models. As reported in [KKT11], graph transformations can be used to collect atomic changes on models into more meaningful user-level changes. In [LWB$^+$13], we presented a domain-specific operation detection approach that transforms model transformation rules into difference patterns. Patterns can then be matched on difference models. In follow-up work, we presented the first search-based approach to detect operation sequences between two model versions without requiring a difference model [bFKLW12, KMW$^+$17]. In our approach, we resorted on atomic difference models in the fitness function to compute how close a computed model is to the given revised model. None of these approaches considered domain-specific distance metrics as we present in this paper. An additional contribution of this paper is the comparison of difference and distance approaches to detect a sequence of operations.

Maoz et al. [MRR11] argue that existing model differencing approaches are purely syntactic and challenge the community to develop semantic difference operators. Two models may be syntactically different but can be semantically equivalent. The authors in [LMK14] presented a semantic differencing approach that reuses the operational semantics of modeling languages. It would be possible to extend our approach to deal with semantic model distances concerned with the instantiation or executability of models.

Finally, there is also related work on using numeric differences of design metrics of models to characterize changes in the context of refactoring detection [DDN00]. For instance, one may compare different versions of class diagrams based on the maximum inheritance depth or number of classes.

6.2 Model Clustering

Some recent works discuss dedicated support to cluster models in model repositories for different purposes, such as providing an overview or performing clone detection. While classical two- or three-way model comparison is mostly studied in the area of model versioning and evolution, model clustering is mostly studied for model repository analytics. This context requires to compare multiple models simultaneously. Thus, the work in [BCVvdB16] proposes a generic model clustering technique. They translate models into a vector space representation to reuse existing generic clustering distance measures. In follow up work [BC17, BCvdB18], the authors have proposed n-grams for clustering of models, again based on the usage of generic distance metrics. In contrast, we propose the usage of domain-specific distance metrics that are directly measured on the model structures. Nevertheless, we also build on established distance measures. However, we configure them for the given modeling languages to incorporate more domain-specific knowledge.

7 Conclusion

In this paper, we have introduced a novel notion of metrics for model comparison, so-called domain-specific distance metrics. We have defined a method to derive tool support to compute these measurements from the metamodel and the change operations of the DSL. We have shown the usage of the domain-specific distance metrics in the recovery of operation-based changes between model versions for different modeling languages.

One main result of our study is that heavily changing model structures seem to be a particular challenge when it comes to the computation of distance metrics. Thus, the cases used in our experiment have shown different performances of calculating and using model distances. As future work, we plan to perform more experiments, with cases such as the refactoring case, where elements are deleted, created, and potentially re-created. Although we have presented three distance metrics, this is by no means complete: we plan to investigate more metrics to characterize the difference between model versions further. Finally, we plan to use model distances for other model management tasks, such as model synchronization and model-metamodel co-evolution, where model differencing approaches are currently used.

About the authors

Eugene Syriani is an associate professor at the University of Montreal. Contact him at `syriani@iro.umontreal.ca`, or visit `http://www-ens.iro.umontreal.ca/~syriani/`.

Robert Bill is a PhD student at TU Wien currently working as researcher for the Austrian Center of Digital Production. His research interest is in the broad field of model-driven engineering with a special focus on language engineering, model integration, and (model) optimization. Contact him at `bill@big.tuwien.ac.at`

Manuel Wimmer is a full professor leading the Institute of Business Informatics - Software Engineering at the Johannes Kepler University Linz. His current research interests are focused on the foundations and applications of model-driven engineering technologies. Contact him at `manuel.wimmer@jku.at`, or visit `https://www.se.jku.at/manuel-wimmer`.

Acknowledgments This work has been partially supported and funded by the Austrian Research Promotion Agency (FFG) via the *Austrian Competence Center for Digital Production* (CDP) under the contract number 854187, by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and by the FWF under the grant numbers P28519-N31 and P30525-N31.

References

[BC17] Önder Babur and Loek Cleophas. Using n-grams for the automated clustering of structural models. In *Current Trends in Theory and Practice of Computer Science*, volume 0139 of *LNCS*, pages 510–524. Springer, 2017.

[BCvdB18] Önder Babur, Loek Cleophas, and Mark van den Brand. Model analytics for feature models: case studies for S.P.L.O.T. repository. In *Proceedings of MODELS 2018 Workshops*, pages 787–792, 2018.

[BCVvdB16] Önder Babur, Loek Cleophas, Tom Verhoeff, and Mark van den Brand. Towards statistical comparison and analysis of models. In *Model-Driven Engineering and Software Development*, pages 361–367. IEEE, 2016.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers, 2017.

[bFKLW12] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search-based detection of high-level model changes. In *International Conference on Software Maintenance*, pages 212–221. IEEE, 2012.

[BP08] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[DAPM02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Object-Oriented Programming Systems, Languages & Applications*, ACM SIGPLAN Notices, pages 166–177. ACM, 2000.

[DRELHE16] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, 2016.

[DRIP12] D. Di Ruscio, L. Iovino, and A. Pierantonio. Coupled evolution in model-driven engineering. *IEEE Software*, 29(6):78–84, 2012.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[FTW16] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-based model transformations with MOMoT. In *Theory and Practice of Model Transformations*, volume 9765 of *LNCS*, pages 79–87. Springer, 2016.

[GKLE13] Christian Gerth, Jochen Malte Küster, Markus Luckey, and Gregor Engels. Detection and resolution of conflicting change operations in version management of process models. *Software and System Modeling*, 12(3):517–535, 2013.

[KDRPP09] D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige. Different Models for Model Matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009.

[KHL^{+}10] Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, and Jörn David. Comparing state- and operation-based change tracking on models. In *Enterprise Distributed Object Computing*, pages 163–172. IEEE, 2010.

[KKPS12] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of model comparison tools. In *Automated Software Engineering*, pages 306–309. IEEE, 2012.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Automated Software Engineering*, pages 163–172. IEEE, 2011.

[KMW+17] Marouane Kessentini, Usman Mansoor, Manuel Wimmer, Ali Ouni, and Kalyanmoy Deb. Search-based detection of model level changes. *Empirical Software Engineering*, 22(2):670–715, 2017.

[KRM+13] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Workshop on Scalability in Model Driven Engineering*, BigMDE'13, pages 2:1–2:10. ACM, 2013.

[LAD+16] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, 15(3):647–684, 2016.

[LMK14] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic model differencing utilizing behavioral semantics specifications. In *Model-Driven Engineering Languages and Systems*, volume 8767 of *LNCS*, pages 116–132. Springer, 2014.

[LWB+13] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[LWG+12] P. Langer, M. Wimmer, J. Gray, G. Kappel, and A. Vallecillo. Language-specific model versioning based on signifiers. *Journal of Object Technology*, 11(3):4:1–34, 2012.

[MRR11] S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *Model-Driven Engineering Languages and Systems*, volume 6627 of *LNCS*, pages 194–203. Springer, 2011.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for EMF model transformation development. In *Graph Transformation*, volume 10373 of *LNCS*, pages 196–208. Springer, 2017.

[SC13] M. Stephan and J. R. Cordy. A survey of model comparison approaches and applications. In *Model-Driven Software Development*, pages 265–277. SciTePress, 2013.

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SPTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, 2001.

[SV13] E. Syriani and H. Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414, 2013.

[TELW14] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications:

From theory to implementation. *Software and System Modeling*, 13(1):239–272, 2014.

[VWV11] Sander Vermolen, Guido Wachsmuth, and Eelco Visser. Reconstructing complex metamodel evolution. In *Software Language Engineering*, volume 6940 of *LNCS*, pages 201–221. Springer, 2011.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962.

[XS06] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Working Conference on Reverse Engineering*, pages 263–274. IEEE, 2006.