Domain-Specific Model Distance Measures

Manuel Wimmer[1], Eugene Syriani[2], and Robert Bill[3]

[1] Johannes Kelper Universität Linz, Austria
[2] Université de Montréal, Canada
[3] Technical University Vienna, Austria

Abstract. A lot of research was invested in the last decade to develop differencing methods for models to identify the changes performed between two model versions. A difference model captures these changes. However, less attention was paid to distance computations of model versions. While different versions of a model may have a similar amount of differences, the distance to a base model may be drastically different. Therefore, we present in this paper distance metrics for models, a method to automatically generate tool support for computing domain-specific distance measures and show the benefits of distance measures over differences in searching for model evolution explanations. The results of running different experiments show... ES ►*todo*◄

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

1 Introduction

With the emergence of model-driven engineering [], the need for dedicated techniques for management models arose. In particular, a lot of research was invested in the last decade to develop differencing methods for models to identify the changes performed between two model versions. Different algorithms for computing differences as well as several representations of differences have been proposed. Difference models which capture the changes between model versions have been used in different cases such as model co-evolution, versioning or synchronization. Thus, difference models represent an important artefact kind in model-driven engineering.

While differences have been in the focus, less attention was paid to distance computations of model versions. Distances may be useful because of several reasons. First, while different versions of a model may have a similar or even same amount of differences, the distance to a base model may be drastically different. Second, distances may be an additional metric to consider the evolution paths of models to reach a certain setting. As an example consider the movement of attributes between different classes. If we have classes A, B, C, D all connected in sequence with an association, and move an attribute from A to another class, we may always have the same difference: attribute deleted in A and added to one of the other classes. However, a distance metric could tell us how far we have moved the attribute by getting a different distance measure if the attribute ends up in class B, C, or D.

In order to provide additional measures for understanding how much a model has changed, we present in this paper distance metrics for models. In addition, we present a

method to automatically generate tool support for computing domain-specific distance measures from an annotated version of the metamodel of the language used for defining the models. Finally, we show the benefits of using distance measures over differences in searching for model evolution explanations. Specifically, the results of running different experiments show...

This paper is organized as follows. In Section 2, we provide an overview on the background of our approach and discuss the present gap between differences and distances. Our approach for generating tool support for computing domain-specific distance measures from metamodels is presented in Section 3. In Section 4, we evaluate the effectiveness of domain-specific distance measures for the particular case of computing model evolution explanations in terms of domain-specific change operations. In Section 5, we survey related work and in Section 6, we conclude with a short summary and an outlook on future work.

2 Background

In this section, we explain the background of our work. As every software artifact, also software models are subject to continuous evolution. Knowing the operations applied between two successive versions of a model is not only crucial for helping developers to efficiently understand the model's evolution (Koegel et al., 2010), but it is also a major prerequisite for model management tasks REF. In general, we may distinguish between two categories of model diffing approaches. The first category describes model diffs as atomic operations, such as additions, deletions, updates, and moves of model elements. The second category uses domain-specific operations (Sunyé et al., 2001) consisting of a set of cohesive atomic operations, which are applied within one transaction to achieve one common goal often expressed as a kind of model transformation. We detail both categories in the following.

2.1 Model Diffs as Atomic Operations

Current model comparison tools often apply a two-phase process: (i) correspondences between model elements are computed by model matching algorithms (Kolovos et al., 2009), and (ii) a model diffing phase computes the differences between two models from the established correspondences. For instance, EMF Compare (Brun and Pierantonio, 2008) – a prominent representative of model comparison tools in the Eclipse ecosystem – is capable of detecting the following types of atomic operations:

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

The advantage of diffing for atomic operations is that generic tool support is available which works for all types of models. On the downside, working with large atomic

differences is challenging and may require a higher level of abstraction. Therefore, the following diffing approaches have been developed over the last years.

2.2 Model Diffs as Domain-Specific Operations

To raise the level of abstraction for model diffs, a more concise view of model differences is required that aggregates the atomic operations into domain-specific operation applications such that the intent of the change is becoming explicit. Existing solutions (Hartung et al., 2010, Küster et al., 2008, Xing and Stroulia, 2006, Langer, Kehrer) provide language-specific operation detection algorithms. Often model transformations are the technique of choice used for specifying executable domain-specific operations. In particular, the operations are specified by transformation rules stating the operation's preconditions, postconditions, and actions that have to be executed for applying the operation. Especially, the approaches proposed by Langer et al. and Kehrer et al. build on model transformations for operation detection between two model versions. The output of these approaches is a set of transformation rule applications which correspond to the list of domain-specific operation applications. By following these approaches, the difference models can be compressed as shown in different studies REF. 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 which evaluate different evolution path with respect to the ability if they can produce the revised version of model. Again, for this atomic differences have been used to compare the computed revised versions with the given revised versions.

2.3 Synopsis

While atomic operations allow to reason about the differences between two models on a fine-grained level, they fail to reason on distances. Assume just the simple Pac-Man example presented in Figure X. The Pac-Man is moving from one area to a very different one in the game field. The same difference is reported for all the intermediate version, namely a change in the assignment of the Pac-Man to the fields. This means, if the Pac-Man just moves to the neighbour field or to the completely other end of the game field, the difference is the same as atomic diffing techniques are fully agnostic about the spatial aspects of elements. Domain-specific operation detection approaches would be more suitable to represent distances as they are able to represent differences as a set of operations. However, for their computation, atomic diffing is required which is not providing an appropriate guidance to decide if the search goes into the right direction or not. For instance, the Pac-Man could move to any other field and we always get the same fitness value of operations sequences. Therefore, we propose the usage of additional distance metrics to provide additional information on top of difference models in the following section and show their benefits for the search of domain-specific operation-based differences in Section 4.

3 Domain-specific model distances

3.1 Running example

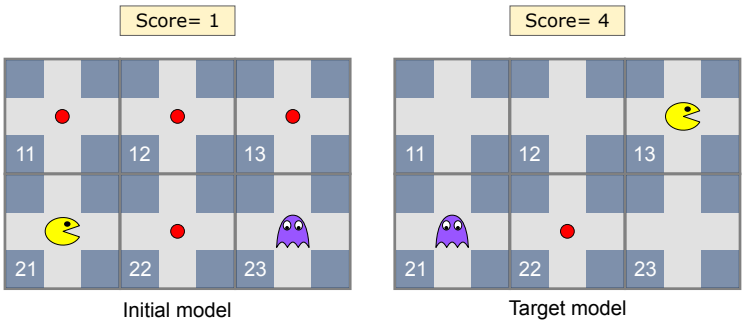

Fig. 1. The initial and target models of a Pacman game configuration

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]. Fig. 1 illustrates two 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. 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 number of food Pacman eats. 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 adjacent grid node. ES ►*should we show one rule in Henshin?*◄ 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 can be applied at any time during the execution of the transformation ES ►*do we need this assumption?*◄ .

Our goal is to find 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 all possibilities 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 the example in Fig. 1, a minimal rule sequence is Pacman moves up once, then moves right three times, each time eating the food. The ghost also moves left twice. ES ►*In current approaches? cite?*◄ To compute the difference between the two models, a generic model comparison tool, like EMFCompare, 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 information is not precise enough to identify the minimal sequence of rules. For example, if Pacman moves right first, then he will have to go through the top center grid node at least twice. Common model difference approaches 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, the notion of Pacman and ghost movements must be encoded in the comparison. Inplace model transformation rules typically encode semantic 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 optimize the search space of identifying the minimal sequence of rule applications.

3.2 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. For our purpose, we need metrics that are 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. These metrics are measured between two models M1 and M2, where M2 is the result of applying a sequence of rules on M1.

The semantics of many formalisms relies on movements of model elements. Some of their elements are *movable* while others represent *positions* where an element can move to. For example, Pacman moves on the grid in our example, attributes move to superclasses in class diagrams, or tokens move between places in a Petri net. Furthermore, some elements are *modifiable* meaning that the transformation changes some of their attribute values, like the score in the rule 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 $Mov, Pos, Mod \subset V$ corresponding to the movable, position, and modifiable objects. In our running example, grid nodes are the position nodes and Pacman and ghosts are movable nodes. Note that, in general, Pos may be different for each $v \in Mov$. Additionally, all three subsets need not to be disjoint nor complete: 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 $N, P \subset E$. 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. The attribute function $a : V \times \Sigma^* \to \mathbb{N}$ assigns numerical values to each attribute name of a node.

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. We define $\delta_M(v_1, v_2)$ as the length of the shortest sequence of neighbor edges connecting v_1 to $v_2 \in Pos$. In Fig. 1, $\delta_M(pacman, grid_{13}) = 3$ and $\delta_M(ghost, grid_{21}) = 2$, which is equivalent to

the Manhattan distance in this grid layout. 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 the move distance between two models as:

$$\Delta_M(G_1, G_2) = \sum_{v_1, v_2 \in Mov_{12}} \delta_M(p(v_1), p(v_2)), \text{ where } l(v_1) = l(v_2)$$

Here, $p(v_1)$ is the position of v_1 in G_1 and $p(v_2)$ is its corresponding position in G_2. In our example, $\Delta_M(M1, M2) = 5$. 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.

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

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

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, $\Delta_E(M1, M2) = \frac{3+0}{13+10} = 0.13$. We can interpret this distance as the ratio of objects added or removed between the two models. In this particular case, 13% of the objects in M1 have been removed in M2. Note that the element distance is not concerned with edges since they are already taken care of by the move distance.

Value distance The third metric is concerned with the difference in attribute values between objects in M1 and M2. We assume that any attribute value can be encoded as a unique number, a common practice for metrics [BMB+18]. We define the value distance of attribute x of node v between G_1 and G_2 as:

$$\delta_V(v, x) = \frac{|a(v, x) - a(v_1, x)|}{a(v, x)}, \text{ where } v_1 \in Mod_1, v \in Mod_2 \text{ and } l(v) = l(v1)$$

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. This distance computes the margin of error needed to obtain the value of x in v from its value in v_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}. This calculates the average margin of error between all attribute values of the two models. In our example, $\Delta_V(M1, M2) = \delta_V(score, value) = \frac{|4-1|}{4} = 0.75$.

3.3 Adapting distance metrics to the DSL

The distance metrics presented in Section 3 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 one of its attribute value is modified in the postcondition of a rule. Such classes define the type of the nodes in Mod.

The three distance metrics rely on the label function l to correspond similar nodes between the two models. For example in Fig. 1, 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. 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.

From the above, we understand that automatically adapting the distance metrics to the DSL is very challenging. Therefore, we implemented the distance metrics as a Java library that can compute 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 his DSL. He must also define the `getId()` function for each class of the metamodel, if it does not already have an identity attribute.

4 Finding a sequence of rule applications

ES ►*as a subsection or a section on its own. This is where we can talk about implementation: MOMot, Henshin, Ecore, etc.*◄

The idea is to generate the optimizing search problem to be tailored for the domain. In this sense, we customize the fitness function based on a distance metric. Multi-objective genetic programming with an objective function to minimize for each distance metric. We also need to minimize the number of rule applications: the number of rules to apply

to get from M1 to M2. This distance is already taken into account in MOMot and returns a positive integer.

5 Evaluation and discussion

This project uses MOMot to search for the sequence of application of model transformation rules that lead an input model M1 to a target model M2. As running example, we use the PacmanGame domain, with the rules specified in Henshin.

Having the evolution recovery problem athand, we apply our search-based framework MOMoT [**?**,**?**], to find the Pareto-optimal module evolutions. MOMoT[4] is a task- and algorithm-agnostic approach that combines SBSE and MDE. It has been developed in previous work [**?**] and builds upon Henshin[5] [**?**] to define model transformations and the MOEA framework[6] 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. In order to apply MOMoT for the given problem, we need to specify the necessary input. 2 model versions, change operators defined as Henshin rules, and the objectives for the search.

Objectives are either based on diff metrics or on distance metrics.

Search based approaches

MOMot - maybe mention MOMoT just in the evaluation section?

All experiments use the same input and output models, the same transformations, the same solution length and the same optimization algorithms and only differ in the fitness functions used in these algorithms. The exact Pacman and Petrinet cases have been freshly created for this evaluation, but are commonly used in the graph transformation world. In the past, the Refactoring case has been used to evaluate different algorithms for MoMOT in the past, where all algorithms optimized the same fitness function. How, this case is used to evaluate the same algorithm with different fitness functions.

5.1 Objective

As our main goal is to compare different fitness functions in terms of their impact on search processes, we want to answer two research questions:

- *RQ1 - Search Space Exploration*: Is there a significant difference in the solutions found by applying each fitness function?
- *RQ2 - Search Time*: Is there a significant difference in the number of iterations required to get good solutions?

[4] MOMoT: `http://martin-fleck.github.io/momot`
[5] Henshin: `http://www.eclipse.org/henshin`
[6] MOEA Framework: `http://www.moeaframework.org`

We answer these research questions by measuring several properties of the final and intermediate results during each experimental run. In particular, to answer RQ1, we compare the final solutions, while we comare the intermediate results to answer RQ2.

5.2 Experiment setup

We evaluate these research questions with three MoMOT case studies. Each case study consists of a single domain model and multiple henshin transformations. The *Pacman* case study has been introduced in Section **??**. The *Petrinet* case study simulates a Petrinet with multiple tokens which can be in different places. A token can be transferred to another place by firing transitions. The *Refactoring* case study, as found in [**?**], is about performing refactoring operations like extracting superclasses or pushing up attributes to a model containing classes and attributes. For each case, we randomly generated multiple test input and output models as outlined in following paragraphs.

	Pacman	**Petrinet**	**Refactoring**
Rule count	High	Low	Medium
Rule complexity	Low	Low	High
Solution length	High	High	Low
Structural changes	No distance changes	-	Distance changes

The case studies have selected as they differ in terms of rule count, rule complexity, and expected solution length as detailed in Table **??**. The Petrinet simulation contains only a single rule which can match in many different ways and whose application does not limit its re-execution. Thus, the rule count is low, but the expected solution length is high. In contrast, the rules for Object Oriented Refactoring typically can be applied in a more limited way and applying it typically limits the application even more, yielding a lower expected solution length. The Pacman example is somewhere inbetween as many rules are defined, but they semantically all do the same, which is moving Pacman or a Ghost, but only differ in what happens on specific fields. Most importantly, only the Refactoring example contains rules which modify the graph in way which matters for the domain specific distance evaluation. A detailed description of the cases follows in the next paragraphs.

Pacman The pacman example was described in the previous sections and is thus not explained further here.

Petrinets This example consists of places which may be connected to other places via transitions. A place can have multiple transitions and each transition can have multiple outgoing states, contributing to a non-determinism in firing the rules. While places and transitions are modeled as named objects, tokens are not objects, but are just modeled as attribute values.

There is only a single rule *fireRule* as shown in Fig. 2 which allows to move a single token from a place to another place. As usual for MoMOT, the parameters of the rule

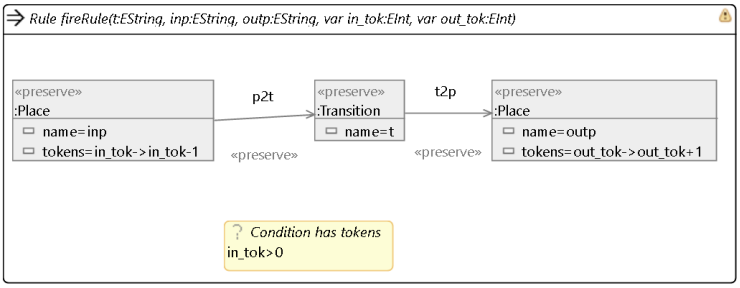

Fig. 2. A transition can fire if there is at least a single token in the source place

uniquely identify the rule match. Even though the model is simple, the number of models generated by applying this rule multiple times can be huge, especially if multiple tokens are used.

Object-oriented refactoring This example [7], which was initially taken from the TTC 2013, contains entities with named, typed properties and generalizations. Originally, it was used as optimization problem to reduce the number of attributes and entities in a system. However, in this paper we want to find a way to refactor a system in a particular fashion. The three refactoring rules are (a) Pull up attribute, which moves an attribute existing in all subclasses into the superclass, (b) extract super class, which creates a new superclass if an attribute is existing in some, but not all subclasses of a particular class, or (c) create root class, which is the same if the classes do not have any existing superclass.

In contrast to both examples above, this example changes the model structure by adding entities and generalizations and removing properties.

5.3 Analysis

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

While many general distance metrics exists, we opted for an EMF-compare based one as this tool is widely used for differentiating models. In particular, we calculate the difference between two models by counting the percentage of model differences in the whole model.

Initially, we generated 100 examples of pairs of source and target model for each case. As the results for the Refactoring case initially were good, but not statistically significant, we generated 1000 examples for this case as well. While the source model was generated randomly within certain predefiend parameters, the target model was generated by randomly applying transformations to the source model. However, for the Pacman case, we had to add more "'intelligent"' rules to generate sensible target states as typically, randomly applying move operations just lead to Pacman being killed soon

[7] Also see `http://martin-fleck.github.io/momot/casestudy/class_restructuring/` for a detailed description

by running into a ghost or just being eaten by a ghost. Here, the rules force Pacman to eat neighboring food when possible and avoiding any ghosts. The search process itself does not use these rules, but just the generic ones.

For each example, only a single run was conducted. Then, the final solutions generated by each run were compared. A solution is considered better if the number of transformations used to reach the target model is smaller. If a run did not produce any transformation sequences matching the target model, this number is considered to be infinite. We did not consider cases where the result models were not exactly matching the target model as at least two different distance metrics could be used for that.

To determine which fitness function was better we assumed that if both fitness functions would produce equal results, there was a 50/50 chance that either fitness function was better for runs which produced differences. For each example, we calculated the chance that the distance metrics would be as good or better just by chance. Thus, we can reject the null hypothesis that the domain specific fitness function is not better if this value is below 5%.

	Generic	Equal	Domain-specific	p-value
Pacman	1	73	27	**.11E-7**
Petrinet	26	30	45	**.016**
Refactor (100)	22	47	32	.11
Refactor (1000)				

Results for RQ1 Search Space Exploration Table 5.3 shows the differences in the results quality for both approaches. For the Pacman case, we generated 8x8 fields with one Pacman, three ghosts, and 15 food, where all food and all other entities were randomly distributed. The Petrinet case used 10 places and 1 to 2 transitions per place and 1 to 2 outgoing place per transition. Also, the net was initialized with between 2 and 4 tokens. The Refactor case used 6 entities, 1 to 5 attributes per entity, where each name was one of 8 possible names and each attribute name was associated to 1 to 3 types. Also, each entity had a 50% chance to have a random superclass.

The Pacman case was rather hard for both fitness functions - all the equal values are due to no solution being found in either case. However, in this instance the domain specific fitness function could show its benefits, as it could solve the problem in 27 cases, while the generic one could only solve it in one case, yielding a significant difference. In contrast, the Petrinet case was rather easy for both algorithms, as most of the equal values were due to both algorithms finding solutions having the exact same quality. Still, the domain specific fitness function allowed us was at least statistically significantly better.

Results for RQ2 Search Time We determined the rate of convergence by storing the solutions found after each evaluation step

While exact time measurements were not taken, the domain-specific distance function always was faster, sometimes drastically faster, than the generic EMF-compare based one.

5.4 Threats to validity

5.5 Discussion

The current implementation assumes that M1 and M2 have the same position objects (none are deleted or created). This is too restrictive. Therefore M1 and M2 should be preprocessed by merging their position elements and then use the move distance. One possibility is to use EMFCompare to merge M1 and M2 based on the position elements.

Interestingly, when you run the Pacman test with models/input12missing.xmi and models/targetNoPac.xmi in one signle run, you don't necessarily get always the same sequence of rule applications. That is because p1 can be killed anywhere along the path of g1.

6 Related work

In this section, we discuss first approaches for model differencing and second approaches which cluster models based on distance metrics.

1) Comparing models. I am only aware of model difference approaches which produce differences and not distances. But I can enumerate here atomic diff approaches and domain-specific operation based diff approaches.

2) Clustering models. Mostly concerned with model repositories. Typical clustering distance metrics based on model features.

`http://ceur-ws.org/Vol-2245/ammore_paper_2.pdf`

`http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220%2f0005799103610367`

7 Conclusion

Summary

Conclusions

Future work

Our approach may be used for model synchronization.

References

BMB+18. Manuel F. Bertoa, Nathalie Moreno, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. Expressing measurement uncertainty in ocl/uml datatypes. In *Modelling Foundations and Applications*, volume 10890 of *LNCS*, pages 46–62. Springer, 2018.

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

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.

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

War62. Stephen Warshall. A theorem on boolean matrices. 9:11–12, 1962.