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

Motivation

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.

Objective

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.

Contributions

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...

This paper is organized as follows. In Section 2, we provide an overview on the prerequisites of our approach and discuss the present gap between specifications for executing composite operations and for detecting applications of them as well as how this gap can be bridged. Our approach for detecting applications of composite operations is presented in Section 3. In Section 4, we evaluate the correctness and completeness of our implementation and investigate the scalability and performance of our implementation in Section 5. In Section 6, we survey related work and in Section 7, we conclude with a short summary and possible extensions of the presented 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. 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. 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.

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 and Kehrer 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.

2.3 Synopsis

While both atomic operations and domain-specific operations allow to reason about the differences between two models on different granularity levels, both fail to reason on distances. Assume just as a simple example a difference in the stored values of an attribute of type Integer. The same difference is reported if the values are mostly equal, e.g., 1 and 0.9999 or totally different, e.g. 1 and 100. Therefore, we propose the usage of additional distance metrics to provide additional information on top of difference models in the next section and show their benefits for a particular use case in Section 4.

3 DOMAIN-SPECIFIC DISTANCES

3.1 Running example

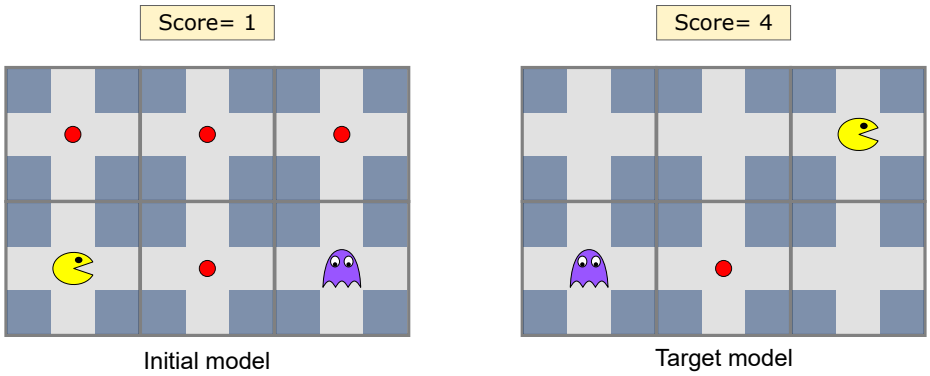

Fig. 1. The initial and target models of the Pacman game configuration

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 positioned 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. 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 represent Pacman moving up, down, left, or right to an adjacent grid node. For similar rules represent ghost movements. ES ►*should we show one rule in Henshin?*◄ Although the rules should obey a certain scheduling, e.g., killing has priority over moving to end the game, in this work we

assume 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?*◄ .

In this work, we want to detect the sequence of rule applications starting from the initial model leading to the target model in Fig. 1. Relying on a search-based technique allow us to infer the sequence of rules by generating intermediate models such that the fitness function is improved. Let us assume that the fitness function uses model difference metrics to decide whether the current sequence of rules will lead to the target model.

Regular difference results: 3 food objects deleted ; 2 on references modified ; 1 score value attribute modified

Distance results: Move distance is $3 + 2 = 5$; Value distance is $\frac{|4-1|}{4} = 0.75$; Element distance is $\frac{3+0}{13+10} = 0.13$;

Minimal rules applied: 1 Pacman Move Up ; 2 Pacman Move Right ; 3 Pacman Eats Food ; 2 Ghost Move Left

3.2 Metrics

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. The metrics to minimize are:

Move distance: There is often movement of elements (e.g., pacman moving on the grid, attributes moving around classes). The move distance of a movable object is the length of the shortest path from its position in M1 to its position in M2. Move distance is related to computing the difference with Ecore references.

Element distance: Is the difference in the presence/absence of elements in M1 and M2. The element distance is the ratio, between 0 and 1, of the number of differences with respect to the total number of objects in M1 and M2.

Value distance: Is the difference in attribute values between M1 and M2. We assume that any attribute type can be encoded as numbers. Then the value distance of attribute x is its margin of error: $|M2.x - M1.x|/M2.x$. In this case, M2 acts as the expected target.

Rule application distance: Is 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.

The *Move distance* relies on the Floyd-Marshall algorithm that computes the shortest from any position element to any position element in an Ecore model. This code is in EcoreShortestPaths and is independent from the domain. Its only external dependency is an interface IEReferenceNavigator which provides a function that gives the neighbor(s) of a position element. DistanceUtil provides all the necessary methods the move distance requires.

The *Value distance* is an average of all attribute distances. We only consider attributes of objects present in both M1 and M2 because the element distance takes care of absence and presence of elements. We assume that any attribute value can be represented as a unique number. DistanceUtil provides the toDouble method to perform that conversion. Currently, it only supports number values encoded as Number or String data types. For each attribute, we compute its margin of error.

The *Element distance* looks for the objects present in M1 but not M2 and M2 but not M1. This is then divided by the size of M1 and M2. We only consider objects instances of metamodel classes. So references and attributes are not taken into account in this measure. This distance relies on the unique ID of each object as returned by the getId method.

DistanceCalculator is the abstract class at the root that should be inherited by your distance function. For example MoveDistance only relies on the move distance between M1 and M2.

3.3 Generation of domain-specific metrics

This implementation is just a proof of concept. It has been implemented with the mindset that the distance calculation is generated automatically from analyzing the metamodel and the transformation rules.

Given a metamodel MM and Henshin rules R, we want to generate the distance calculator that will be used by the MOMot script.

The domain-specific distance classes (e.g., the move, element, value distances) can be easily generated. They have two dependencies to the metamodel:

The package instance, by overriding the getEPackageInstance() function The constructor, by instantiating the appropriate DistanceUtil singleton object specific to the metamodel

Only the utility class (e.g., PacmanDistanceUtil) must be generated after analyzing MM and R. Following the code in PacmanDistanceUtil should guide you to know how to generate the code. Here are special considerations:

Your Utility class must inherit from DistanceUtil. It should import all movable, position, modifiable, and other classes. It must provide a function used for its singleton instantiation as follows.

It should have 4 attributes for movable, position, modifiable, and all other types.

It should override all abstract methods from DistanceUtil. The tricky part is the generation of the Object getId(EObject object) method. If each object has attribute with setID(true) you can rely on super. Otherwise, you have to make up one unique on your own. For example, if you know for sure that an attribute is unique, then you can rely on it (e.g., Places and Transitions in the Petrinet example). If there is only one possible instance of this element, then return true (e.g., Scoreboard in the Pacman example). It can also rely on an object it references or that references it (e.g., Food in the Pacman example).

You also need to generate the DistanceUtilFactory specific to the metamodel (e.g., PacmanGameDistanceUtilFactory). Its only purpose is to make the concrete DistanceUtil accessible as a singleton.

Customization The only code that is specific to the metamodel is the class that implements DistanceUtil. This is where you define the functions that provide the following information:

The movable objects:	An object is movable if, when analyzing the rules, it has a reference to a position object and the rules modify that reference. Note that it could also be that a position object references a movable object.
The position objects:	An object is a position if, when analyzing the rules, it is what movable objects are always linked to. Note that it could also be that a position object references a movable object.
The modifiable objects:	An object is modifiable if, when analyzing the rules, one of its attributes changes value.
The other objects:	Any object that is not movable, position or modifiable.
The ID of an object:	the value that uniquely identifies an object. This is used to find similar elements between M1 and M2.
The modifiable attributes:	All attribute values subject to modification for a given object.
Accessing the position:	the attribute used to know the position of a movable object.
ccessing the neighbors of a position:	the attribute used to connect position objects.
Accessing the root:	used to find the root object of M1 and M2.

4 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

[4] MOMoT: `http://martin-fleck.github.io/momot`

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

[6] MOEA Framework: `http://www.moeaframework.org`

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.

4.1 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?

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.

4.2 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.

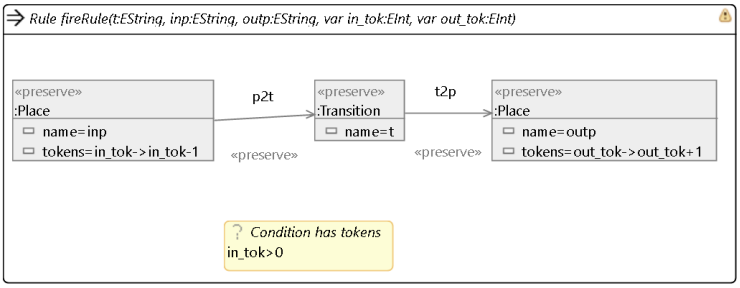

Fig. 2. A transition can fire if there is at least a single token in the source place

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 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.

4.3 Analysis

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

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

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 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 4.3 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.

4.4 Threats to validity

4.5 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.

5 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`

6 CONCLUSION

Summary

Conclusions

Future work

Our approach may be used for model synchronization.

References

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