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

Domain-Specific Model Distance Measures

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

a. Université de Montréal
b. University of Vienna
c. University of Linz

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...

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

Pacman

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.

3.3.1 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.
ng 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[1] is a task- and algorithm-agnostic approach that combines SBSE and MDE. It has been developed in previous work [?] and builds upon Henshin[2] [?] 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. 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?

4.1 Objective

Research questions. Compare with EMFCompare.

[1]MOMoT: `http://martin-fleck.github.io/momot`
[2]Henshin: `http://www.eclipse.org/henshin`
[3]MOEA Framework: `http://www.moeaframework.org`

4.2 Experiment setup

Pacman
Petri nets
OO refactoring

4.3 Analysis

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

TODO
model diff models
model repository: model clustering

6 CONCLUSION

Summary
Conclusions
Future work

References

About the authors

Manuel Wimmer is a ... in ... at .... Contact him at `EMAIL`, or visit `URL`.

Eugene Syriani is an associate professor in the department of computer science and operations research at Université de Montréal. Contact him at `syriani@iro.umontreal.ca`, or visit `www.iro.umontreal.ca/~syriani`.

Robert Bill is a ... in ... at .... Contact him at `EMAIL`, or visit `URL`.

Acknowledgments This work has been sponsored by