Carl von Ossietzky
Universität Oldenburg

Bachelorstudiengang Informatik

BACHELORARBEIT

General Model Difference Calculation

vorgelegt von
Christoph Alexander Küpker

Betreuender Gutachter
Prof. Dr. Andreas Winter

Zweiter Gutachter
Dilshodbek Kuryazov

Oldenburg, 10.06.2013

Abstract
Models, used to represent abstract views on a complex system, are commonly used design documents and artefacts in the process of software development. Differences between consecutive versions of models allow, for instance, model evolution investigation or versioning of models. Such differences, referred to as modelling deltas, are detected using model difference calculation algorithms. Currently available tools and frameworks for model difference calculation are limited either in model language capabilities or have restricted functionalities for implementational reasons. Hence, general model difference calculation is an open problem.
Existing algorithms, tools and frameworks for model comparison have been investigated and a general algorithmic solution has been found. That algorithm has been implemented in a prototype called GDiff which allows model difference detection in a generic manner. The implementation is capable of detecting differences for any model type or language. This is validated by exemplary comparisons of UML Activity Diagrams and a domain-specific graph type used to represent object-oriented software systems.
The solutions to a general model difference calculation algorithm and the prototype implementation show that model difference calculation is actually possible without restrictions to model type or language. The solution provided in this work could for example be embedded in a model versioning system, allowing the system to make models from any language versionable without requiring specific modelling tools or model representations.

Contents

List of Figures

1. Introduction

Versioning is a key component of collaborative software development and evolution. Focused on coding, source code management tools like CVS [4], Subversion [5] and Git [6] allow software developers to store all the incremental versions of a software system. This allows backtracking of development and makes software evolution and collaborative development traceable.
The same concept could be useful for modelling in software development and evolution in general, as well as in model driven software development methods like the Model Driven Architecture (MDA) [7] approach. For any of these aspects, models are created to allow abstract views of the developed or evolving system for different stakeholders. These models are, just like the system itself, updated and changed throughout the development process and are thereby evolving through different versions.
Versioning systems that are based on text documents, such as the previously mentioned source code management tools CVS, Subversion and Git, could be used for the versioning of models if they were transformed into a textual representation, for example, XML Metadata Interchange (XMI) [8] format or a proprietary, text-based model storage format. That is not a suitable solution because it hides the modelling semantics of the processed models from the versioning system and the user. Different versions of a textually represented model would, for example, differ in line additions or character changes. Users would however not be able to trace changes made to the models directly or display model differences on model level.
Some of the UML modelling tools, especially Visual Paradigm [9], provide special versioning systems for created models. These, for example the teamwork client plugin in Visual Paradigm in collaboration with the Visual Paradigm Teamwork Server [10], store the versioned models inside a source code management system repository, for example the one provided by Subversion, adding model versioning functionalities on top which keep the modelling semantics available. Modellers can for example display changes between versions on model level. But this approach binds modellers to one specific modelling tool and thereby restricts versionable modelling languages and model types to the ones provided by the used tool.
Therefore none of these approaches, neither source code management tools nor modelling tool's collaborative plugins, provide modellers with the versioning functionality needed to handle arbitrary modelling projects. Thus, a model-focused but model type and modelling language independent versioning system is needed.

1.1. Goal

Versioning systems do not store the complete model in all its successive versions but instead store the differences between them. Every version can be recreated by transforming these differences into specific versions of the object, for example by applying a specific difference to a specific version to generate the succeeding version. All the activities inside a versioning system that provide the versioning functionalities are handling such differences.

This work focuses on providing a general solution for the most basic activity, the *difference calculation*. Without this basic activity none of the other fundamental versioning activities such as the transformation of objects to previous versions or the visualisation of differences would be possible. Hence, difference calculation builds the foundation for a versioning system.

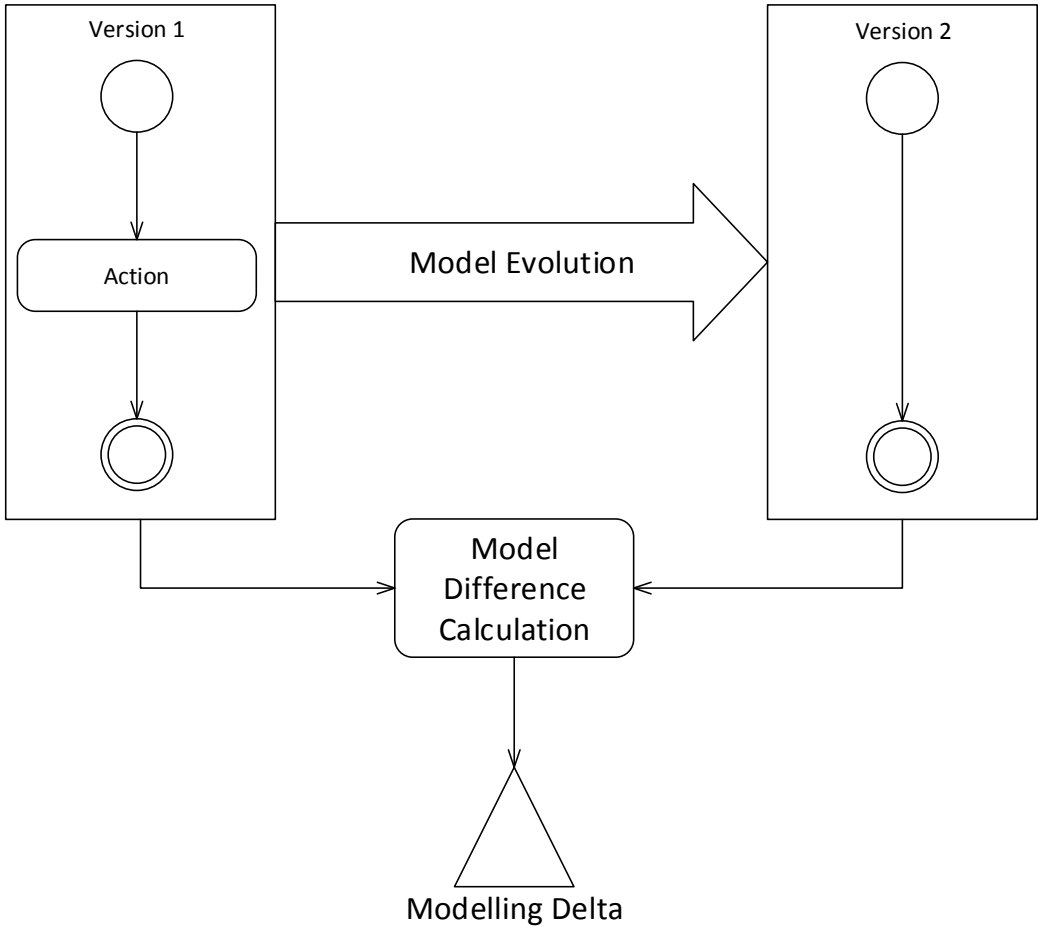

Figure 1: Modelling delta calculation workflow from two successive versions of an UML Activity Diagram

A directed difference between two models is being referred to as the *modelling delta* whose creation is sketched in figure 1. In this case the difference between two successive versions of an UML Activity Diagram is being calculated. The modelling delta contains information on how to modify a specific version of a model to obtain another specific version of the same model for example by listing elements that have to be added, removed or changed. A delta, in contrast to a difference, is directed from a specific host

version to a specific target version of the same model and is generally not bidirectional. A difference calculation algorithm needs to identify these changes in the model versions and output them in a generic delta format.
A variety of model difference calculation algorithms has been developed over the last years, for example SiDiff [2, 11], UMLDiff [12] or DSMDiff [13], but each of these is somehow limited in its approach or its model handling capabilities. Hence, there currently is no general solution to model difference calculation. Therefore, the main goals of this work are:

1. Find a general algorithmic solution to model difference calculation.
2. Implement the algorithm so that any model type and language is generically processable.

The term generality is in this case defined as the capability to handle any model without requiring any model-specific or even user-set configuration as well as the generated difference representation being generated without any further configuration.

1.2. Approach

First an algorithmic solution to graph difference calculation is needed. Related work, for example the frameworks and tools mentioned before, will be investigated to find suitable algorithms providing the needed results. The investigation will be focused on runtime complexity and difference detection correctness. So that even large models, as created for large software systems, can be processed in appropriate time.
Once a problem solving algorithm has been identified, it will be implemented in a tool that allows model difference calculation for any model type and language.
This thesis begins with a description of a Model Versioning System in section 2 which outlines the system boundaries for a model difference calculation algorithm and results in its requirements. Section 3 contains a theoretical investigation of existing model difference algorithms and section 4 describes the idea and the prototype implementation of the resulting model difference calculation algorithm GDiff. Section 5 contains the validation of GDiff against UML Activity Diagrams and other model types and the thesis ends with a conclusion in section 6.

2. System Boundaries and Requirements for a Model Difference Calculation Algorithm

The goal of this work is to find a general solution to model difference calculation independent from the modelling language and type of the processed models. This section describes the embedding for such an algorithm and outlines the system boundaries and functionalities of a Model Versioning System (MVS) within which it can be used.
Examples used to explain the process of model difference calculation and underlying methods are based on UML Activity Diagrams [14, pp.319-434]. For simplification of the examples, a subset of UML Activity Diagrams is used for which the metamodel is given in figure 2. Activity Diagrams thereby consist of Activities which can contain

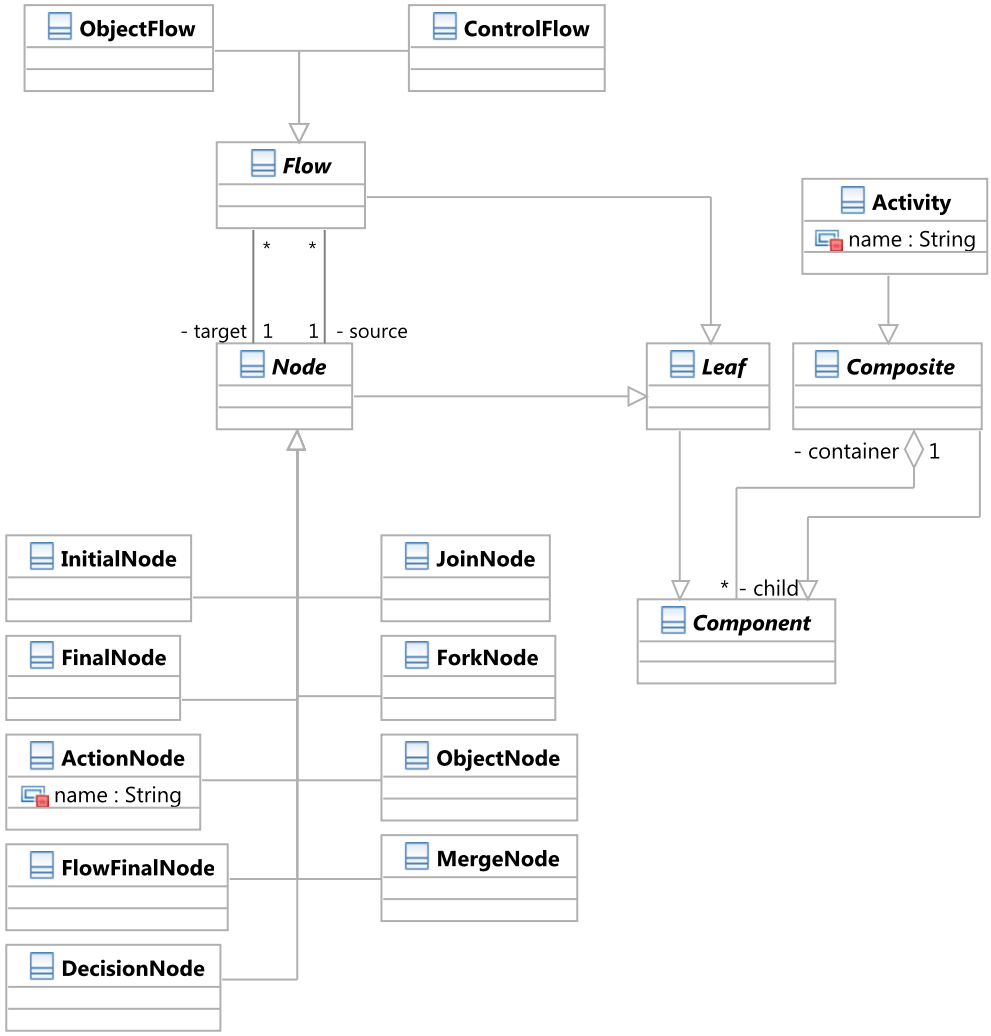

Figure 2: Reduced metamodel for UML Activity Diagrams

other Activites and Nodes, represented by a composite pattern. Nodes can be of type Initial, Final, Action, Decision, Fork, Merge Join, Flow Final or Object Node. Nodes and Activities can be connected using flows, which are directed associations either of type Control or Object Flow. Hence, the metamodel contains the basic concepts of UML Activity Diagrams.

2.1. Model Versioning System

An MVS can be used to assist collaborative modelling projects where multiple modellers work on the same modelling project to, for example, design a software system. A figure of the collaborative modelling process is given in figure 3.

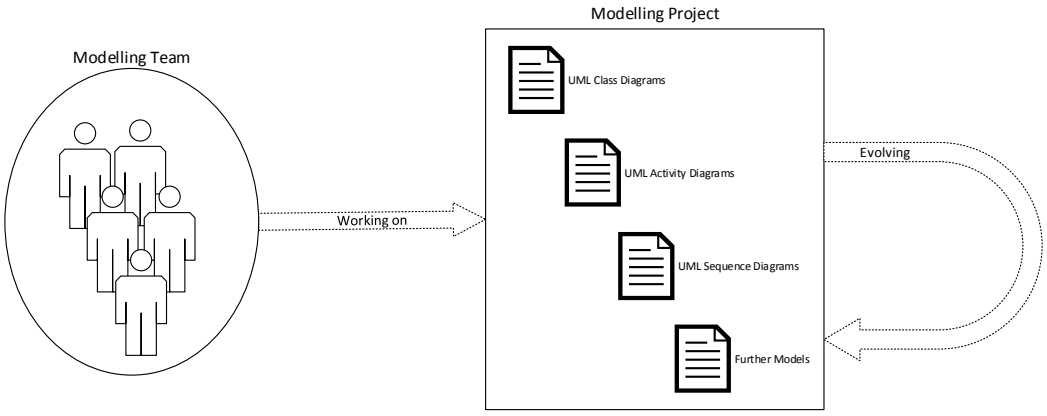

Figure 3: Collaborative Modelling

The modelling project consists of various models of equal or different types which represent certain aspects of the designed system. Each modeller can be assigned to specific aspects but it is also possible that modellers work on the same model at the same time. Whenever team members apply changes to a model, the modelling project evolves to a new version. An MVS allows the team members to store the incremental versions of the project in a centralised repository provided by the MVS. Earlier versions of the project can be retrieved from the repository and new versions of the project can be stored without overwriting previous versions. Additionally, the MVS is capable of conflict handling. Conflicts occur, for example, whenever two or more team members make different changes to the same model at the same time meaning that the changes have been applied to the same version of the same model. The MVS detects such conflicts and provides techniques to solve them, for example, by trying to merge both changes into the model or, if that is not possible, by allowing the user to merge conflicted models by hand.
This functionality is comparable to the ones provided by source code management systems such as Subversion [5] or Git [6], but instead of their text based versioning capa-

bilities, an MVS handles models on model level abstracting from their, possibly textual, internal representation.

2.2. Generic Model Versioning System

For the embedding of a general difference calculation algorithm the Generic Model Versioning System (GMoVerS) as proposed by D. Kuryazov et.al. [15, 16, 17] forms the system boundary.
GMoVerS provides the functionalities of a versioning system as described in the previous section for models in general. It aims to be modelling language independent and provides versioning functionalities for any model.

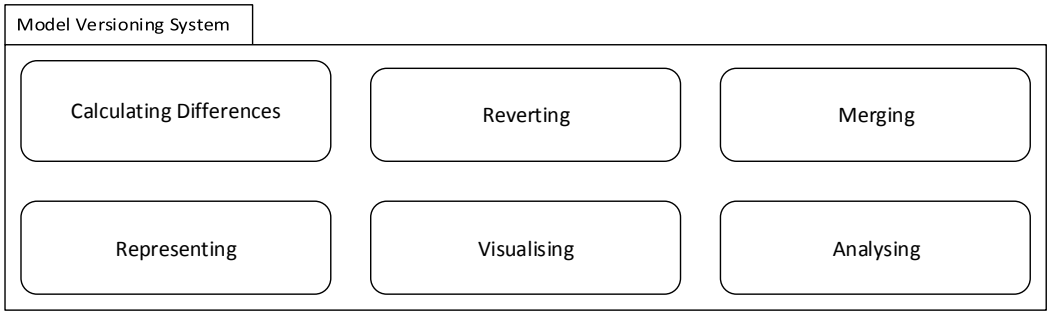

Figure 4: Activities of a Model Versioning System

The MVS consists of six basic activities, shown in figure 4, that in combination provide the desired functionalities of an MVS.
The *calculation of differences* combined with the *reverting* of models allow the MVS to store and retrieve all versions of a model without having to store each model completely. Therefore, model differences are the main artefact used by the MVS. The general workflow of using model differences in an MVS is given in figure 5.
The difference calculation activity, preferably a standalone tool in the MVS, detects differences between two consecutive versions n and $n+1$ of the same model. The metamodel is used to get information about the general structure of the processed models. The resulting output is used in the reverting activity, also preferably a standalone tool, to apply the detected differences to one of the original models and thereby generate the other version of the same model. In figure 5 the reverting activity produces the version n of the model by receiving the model difference and version $n+1$ of the same model. Again, the metamodel might be used to get information about the general structure of the processed model. By relying on this workflow, the MVS can easily keep track of all model versions without needing to store them completely.
The *merging* activity handles conflicts whenever two or more modellers made changes to the same version of a model at the same time. Such conflicts are handled either by providing automatic conflict solving algorithms, which result in a merged version of

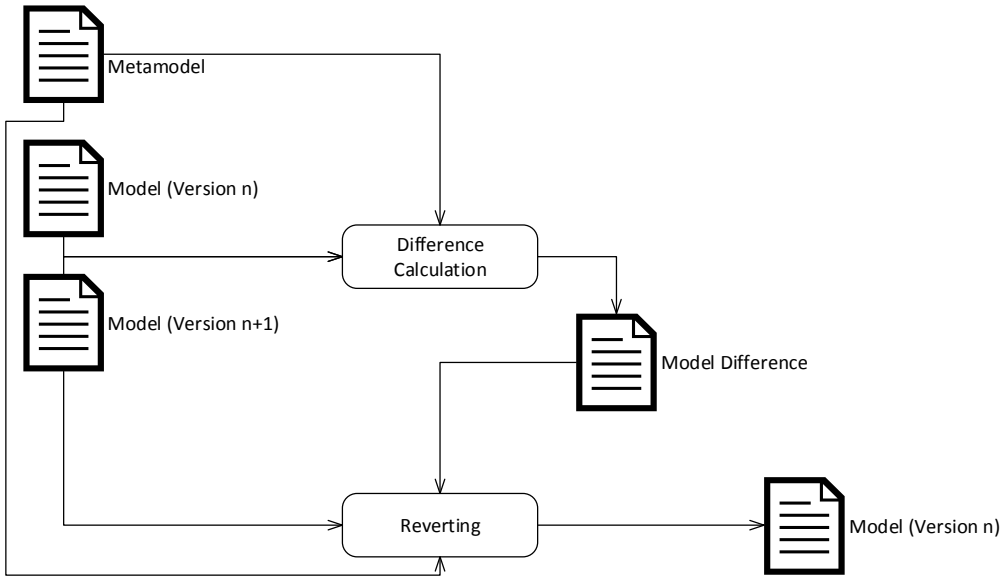

Figure 5: Model difference calculation and reverting workflow

both changes, or by letting the users of the MVS merge their changes by hand. Model differences are *represented* in an MVS-specific delta description language. This difference representation is generated by the difference calculation and used by all activities using model differences. Differences can also be visualised using the *visualising* activity so users can directly see applied changes on model level, for example, by highlighting added, changed or deleted elements in different colors. The *analysis* activity provides model evolution investigation functionalities.
Because model differences are the main artefact within the MVS, the model difference calculation algorithm researched in this work will be the foundation for any other activity of the MVS. But, as the generated output of that algorithm should be in the format of the MVS's delta representation format, this format has to be defined before.

2.3. Modelling Deltas

The differences between models are represented by using so-called *modelling deltas* which are a directed representation of the model differences. The distinction between a model difference and a modelling delta can be explained using natural language.
The two UML Activity Diagrams given in figure 6 are consecutive versions of the same model with version 2 being the most recent version. The difference between these two models can be naturally expressed by saying: "The action node is missing." This information however does not allow the generation of the second version of the model by only knowing the first version and the described difference. Therefore, modelling deltas, even if represented in natural language, contain more specific and especially directed informa-

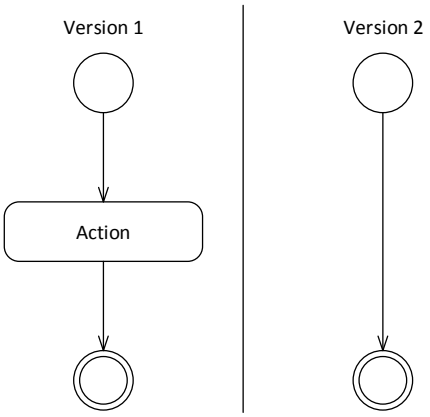

Figure 6: Two consecutive versions of an UML Activity Diagram

tion on how to obtain the second version of the model. A natural language example of a delta for the given example would be: "To generate version 2 of the model from version 1 of the model, remove the action node, the control flow outgoing the action node, and redirect the remaining control flow's target to the final node." This information is complete and allows the retrieval of version 2 by only knowing the given delta and version 1 of the model. With this information given, the reverting activity can generate any model version stored in the MVS.
Due to the fact that modelling deltas are directed, they are not bidirectionally applicable. The delta given before is referred to as a *forward delta*, because it is directed to the newer version of the model. The opposite is referred to as a *backward delta* which would be directed from version 2 to version 1. It is obvious that the backward delta is the inverse of the forward delta. The natural language representation of this delta would be: "To generate version 1 of the model from version 2 of the model, add an action node with the name Action, redirect the existing control flow's target to the new action node, and add a control flow going from the new action node to the final node."
A formal notation for forward and backward deltas is given by the equations

$$m_1 + \Delta_{1,2} = m_2$$
$$m_2 + \Delta_{2,1} = m_1$$

where m_1, m_2 are the models in version 1 and 2, $\Delta_{1,2}$ is the forward delta leading from version 1 to version 2, and $\Delta_{2,1}$ is the backward delta leading from version 2 to version 1. The +-Operator is defined as the application of a delta to a model. This formal representation of delta application and delta direction representation will be used throughout this thesis.
Because a natural language delta representation is hardly processable by automatic tools, a more detailed and linguistically restricted delta description language should be used to represent deltas. That representation can be used by all MVS activities which handle

modelling deltas, such as the reverting activity, without requiring delta transformations.

2.4. Delta Description Language

The differences expressed in a delta can be categorised into additions, removals and changes of elements. By having these, all possible changes a model has gone through can be expressed. Several approaches to model difference representation exist [18, 11]. However, due to the fact that the target of this work is to embed the resulting algorithm in the MVS described before, the operation-based delta description language [16, 17] used in that system is used to represent modelling differences.
The delta description language provides a simple set of instructions that can be used to generate another version of a model. These operations are, just like the difference categorisation given before, categorised into additions, removals and changes. The operations are named according to the metamodel of the processed model and are given in the context of complete models or in the context of an existing model element. The sequential execution of the operations given in a delta can be used to generate another version of a model.

2.4.1. Additions

To create a new element in the target version of the model, an *add* operation is added to the delta. This operation defines the type of the element and assigns the resulting element to a variable that can later be referred to. An exemplary use of the operation is

```
g1 = addActionNode("Name");
```

where `g1` is the variable to which the element is assigned and `ActionNode` is the name of the type of the created element as defined by the metamodel. Attribute values are directly passed as parameters, such as the name of the created Action Node in the example above, ordered according to the order of attributes in the metamodel. The add operations for other element types are named accordingly.
As elements can be contained within other elements, additions of elements to other elements also have to be possible in the delta description language. This is achieved by referring to the desired container via a dot-notation as in

```
g1 = g0.addActionNode();
```

where `g0` is the delta variable of the desired container. In the case of UML Activity Diagrams this could for example be an Activity.
When adding relations between two existing elements, source and target of the created relation must be defined within the add operation which changes the add operation to

```
g2 = addControlFlow(sourceNode,targetNode);
```

Again, just like when creating standalone elements, the resulting element is assigned to a variable named `g2` and the type of the element is given by the operation name. The parameters `sourceNode` and `targetNode` must however be defined elsewhere in the delta so that they actually point to the correct source and target model elements.

2.4.2. Changes

Change operations are used to change attribute values of elements. These operations can be called on any element that has previously been assigned to a variable, for example by using a add operation. To change the name attribute of an action node which is assigned to the variable `g1` to the new value "Action", the operation

```
g1.changeActionNodeName("Action");
```

is added to the delta. Again, the type of the changed element as well as the name of the attribute are given by the operation name and are conforming to the names within the metamodel. The parameter defines the new value of the attribute. The "."-notation, similar to object-oriented programming languages, allows the execution of operations on unique elements.
Because source and target elements of relations in a model can also be expressed using attributes, the same change operations are used to redirect existing relations. But instead of setting a parameter with the type of the changed attribute, the parameter must be another model element that has been assigned to a variable elsewhere in the delta:

```
g.changeControlFlowSource(newSource);
g.changeControlFlowTarget(newTarget);
```

The change of a containment can be expressed by changing the Container attribute of an existing element as in

```
g.changeActionNodeContainer(container);
```

where `container` is the delta variable referring to the desired container.

2.4.3. Deletions

Deletions are again called on elements that have been previously assigned to a variable name. The *delete* operation syntax is

```
g.deleteActionNode();
```

where `g` is the assigned variable and `ActionNode` is the type of the removed element.
The forward delta $\Delta_{1,2}$ resulting from a difference calculation of the models given in figure 6 using this delta description language contains the following operations:

```
g2.deleteActionNode();
g4.deleteControlFlow();
g5.changeControlFlowTarget(g3);
```

The backward delta $\Delta_{2,1}$ contains the following operations instead:

```
g2 = addActionNode();
g2.changeActionNodeName("Action");
g4.changeControlFlowTarget(g2);
g5 = addControlFlow(g2,g3);
```

Important is the fact that multiple deltas representing the same difference are possible. This applies for a natural language delta representation as well as for the used delta description language. For example, it makes no difference which one of the control flows

is redirected and which one is deleted in the forward delta, as long as one is removed and one is redirected.
These deltas are however incorrect. As stated before, the variables used must be defined within the delta document so that they refer to an existing or created element. This is not the case for both of the deltas given above due to the fact that the delta description language does not define how an existing element can be uniquely selected and assigned to a variable other than using an add operation. To solve this issue a deeper investigation of the consecutive version delta creation is necessary.

2.5. Variable Assignment in the Delta Description Language

In figure 6 two consecutive versions of an UML Activity Diagram are given. That example is extended by two more versions of the same diagram which are shown in figure 7. The first version of the diagram is now an empty model whilst the previously existing versions are now referred to as version 2 and 3. Additionally, version 4 of the diagram has been added where the action node has been re-added and the diagram is extended by a fork and a merge node.

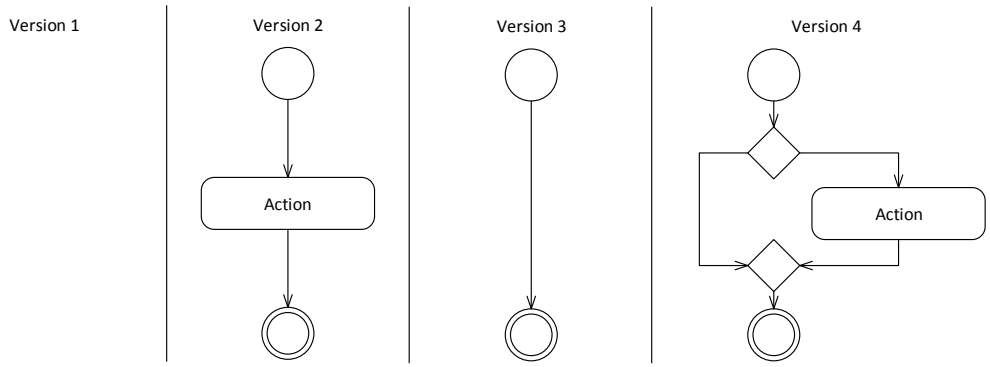

Figure 7: Four consecutive versions of an UML Activity Diagram starting with an empty diagram

The solution to unique and distinguishable variable names within the delta is based on the idea of *creation deltas* that are solely used to create a specific version of a model. These deltas can be applied to an empty model to generate the desired version of the model and are therefore formally referred to as $\Delta_{\emptyset,n}$ where n is the desired version of the model. This idea is explained using the above example and by theoretically generating all forward and backward deltas between these consecutive versions.

2.5.1. Forward Delta

The first difference to calculate is the difference between versions 1 and 2. Because version 1 is the empty model, the delta will only contain add operations. Because all the

elements in version 2 are newly created, the referral to previously existing elements is not a problem. Additionally, the new elements will immediately have variable assignments due to the fact that the add operations of the delta must assign the elements to new variables. By providing a starting number for variable numbering to the difference calculation, it can additionally be guaranteed, that each variable name has never been used before. In this case, the numbering would start at 1 so that the first element added will receive the variable name `g1`. By using these preliminaries, the resulting delta $\Delta_{1,2}$, which is equal to the delta $\Delta_{\emptyset,2}$, contains the following operations:

```
g1 = addInitialNode();
g2 = addActionNode("Action");
g3 = addFinalNode();
g4 = addControlFlow(g1,g2);
g5 = addControlFlow(g2,g3);
```

The difference between version 2 and 3 now somehow has to refer to elements that are already existing. If the previous delta is provided to the difference calculation tool, it can easily be detected that the action node to be removed has previously been added as an element assigned to the variable `g2`. The same conclusion can be applied to any other element that has to be referenced. This leads to the resulting delta $\Delta_{2,3}$:

```
g4.changeControlFlowTarget(g3);
g5.deleteControlFlow();
g2.deleteActionNode();
```

When computing the difference between versions 3 and 4, the variable names for elements which already exist in version 3 cannot be concluded because there is no delta $\Delta_{\emptyset,3}$ given. However, because whenever the newest version of a modelling project is retrieved from the MVS, for example when a team member checks out the current state of the project, the retrieval of the newest version of a model might be the most common case. The complete storage of the newest version n of a model therefore seems adequate. But, due to the fact that the delta $\Delta_{\emptyset,n}$ also helps creating the delta it makes sense to store this delta instead of the newest model itself. This also has the advantage that the MVS only has to handle deltas internally because every model is internally represented as a delta. The delta $\Delta_{\emptyset,3}$ has not been given in the previous step of the given example but as it is needed to compute the difference for succeeding versions, it also has to be generated while calculating the difference between versions 2 and 3. The elements in version 2 have already been assigned to unique variables and it is undesirable behaviour to change these assignments when generating the delta $\Delta_{\emptyset,3}$ because it would make backtracking of model element evolution impossible. Therefore, the generated delta contains add operations which assign the elements to their previously given variables:

```
g1 = addInitialNode();
g3 = addFinalNode();
g4 = addControlFlow(g1,g3);
```

If $\Delta_{\emptyset,3}$ is known when calculating $\Delta_{3,4}$ the variables of already existing elements, such as the initial and final node, can easily be derived. Because the last newly assigned variable occured in $\Delta_{1,2}$ with the unique numbering value 5, the numbering for newly

added elements in version 4 starts at the value 6 to guarantee unique variable names for all versions:

```
g6 = addDecisionNode();
g7 = addActionNode("Action");
g8 = addMergeNode();
g9 = addControlFlow(g6,g7);
g10 = addControlFlow(g6,g8);
g11 = addControlFlow(g7,g8);
g12 = addControlFlow(g8,g3);
g4.changeControlFlowTarget(g6);
```

$\Delta_{\emptyset,4}$ also has to be generated for consecutive versions and contains the following operations, again using previously assigned variables only by knowing $\Delta_{\emptyset,3}$:

```
g1 = addInitialNode();
g3 = addFinalNode();
g4 = addControlFlow(g1,g6);
g6 = addDecisionNode();
g7 = addActionNode("Action");
g8 = addMergeNode();
g9 = addControlFlow(g6,g7);
g10 = addControlFlow(g6,g8);
g11 = addControlFlow(g7,g8);
g12 = addControlFlow(g8,g3);
```

2.5.2. Backward Delta

The generation of backward deltas does not change the order of difference calculation but instead changes the direction of the generated deltas. Just like before, the first version comparison occurs between versions 1 and 2. The resulting delta is the backward delta $\Delta_{2,1} = \Delta_{2,\emptyset}$ which in this case would only contain delete operations. But as the elements in version 2 have never been assigned to variables before, they have all been added in version 2, $\Delta_{\emptyset,2}$ is needed to have these variable assignments. But, as said before, this delta again can be used for retrieval of the newest version of the model within the MVS. The unique element numbering again starts at 1 which results in $\Delta_{\emptyset,2}$ being equal to the same delta given before when investigating forward deltas. Thereby, $\Delta_{2,1}$ contains the following set of operations:

```
g1.deleteInitialNode();
g2.deleteActionNode();
g3.deleteFinalNode();
g4.deleteControlFlow();
g5.deleteControlFlow();
```

By knowing $\Delta_{\emptyset,2}$, the delta $\Delta_{\emptyset,3}$ which is needed for variable assignment and model retrieval, can easily be derived. It contains the same operations that were in $\Delta_{\emptyset,3}$ when investigating forward deltas. The backward delta $\Delta_{3,2}$ then contains add operations which assign the elements to the variables already used in $\Delta_{\emptyset,2}$:

```
g2 = addActionNode("Action");
g5 = addControlFlow(g2,g3);
```

```
g4.changeControlFlowTarget(g2);
```

For comparison of version 3 and 4 the delta $\Delta_{\emptyset,4}$ is created using variable assignments from $\Delta_{\emptyset,3}$. The unique numbering for new elements that did not exist in version 3 starts at 6. It contains the same operations that were in $\Delta_{\emptyset,4}$ when investigating forward deltas. The backward delta $\Delta_{4,3}$ now contains delete operations for elements added in version 4:

```
g6.deleteDecisionNode();
g7.deleteActionNode();
g8.deleteMergeNode();
g9.deleteControlFlow();
g10.deleteControlFlow();
g11.deleteControlFlow();
g12.deleteControlFlow();
g4.changeControlFlowTarget(g3);
```

Concluding from this backward delta investigation, whenever a newest consecutive version n is processed, a virtual empty model is assumed to follow that model version to allow a backward delta calculation which generates $\Delta_{\emptyset,n}$. This specific backward delta assigns all variables needed for $\Delta_{n,n-1}$ and additionally allows a direct retrieval of the newest version of the model from the MVS by simply executing

$$\emptyset + \Delta_{\emptyset,n} = m_n$$

and returning the resulting m_n. This makes it obsolete for the MVS to handle any model representation other than its own delta representation.

2.5.3. Conclusion

Seeing that both approaches, forward as well as backward delta generation, solve the possible problem of variable assignment, the decision about delta direction can solely be based on the aspects of model retrieval. As said before, the most common case is the retrieval of the newest version n of the model. It therefore makes sense to store the delta leading to this newest version, formally $\Delta_{\emptyset,n}$. The version $k < n$ of the same model can then be generated by applying a concatenation of consecutive backward deltas to an empty model:

$$m_k = \emptyset + \Delta_{\emptyset,n} + \Delta_{n,n-1} + ... + \Delta_{k+1,k}$$

Another advantage of using creation deltas within an MVS, is the possible improvement to calculate differences between creation deltas instead of models. Comparing the creation deltas $\Delta_{\emptyset,3}$ and $\Delta_{\emptyset,4}$ reveals some text based differences. These differences can actually be used to derive the backward delta $\Delta_{4,3}$ as well as the forward delta $\Delta_{3,4}$. However, due to the fact that this thesis aims for a solution to model difference calculation, such a delta difference calculation is out of this work's scope.

2.6. Requirements for a Model Difference Calculation Algorithm

The functionalities of a model difference algorithm have been described in sections 2.3 to 2.5 but the requirements for the algorithm, respectively a model difference calculation

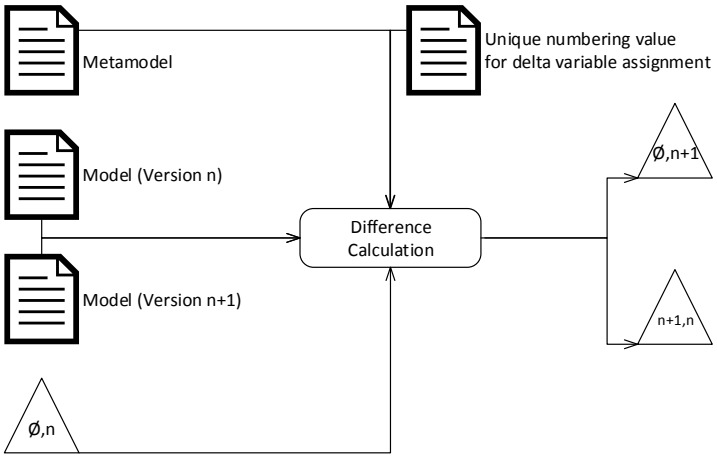

Figure 8: In- and output data for difference calculation tool

tool, have still to be defined.
The algorithm should generally process two models and output the delta in the described format. Because of the used delta description language and the resulting variable assignment investigations in the previous section, the in- and output parameters for the algorithm have to be changed from the ones given in figure 5. The new in- and output parameters are given in figure 8.
Instead of only receiving two consecutive versions of the same model and the metamodel to which these models conform, the algorithm also needs the creation delta which generates the older version of the model. The information contained in these four input parameters is then used to generate two deltas: A creation delta $\Delta_{\emptyset,n}$ which generates the newest version of the model and a delta $\Delta_{n+1,n}$ representing the difference between the two given versions of the model. To allow unique and distinguished variables within these created deltas, a unique numbering value that can be used to create new variable names also has to be provided to the algorithm.
As the algorithm shall be capable of handling any modelling language and model type, the processed models must in some way be generalised so that the algorithm has not to be adjusted to the processed model in any way. To achieve this, some MVS provided external preprocessing component might be necessary. That component would be used to transform the internal model representation of the MVS into the model representation used internally by the model difference calculation tool. This internal representation however must represent models in such a generic way, that it allows the algorithm to actually handle any modelling language without further configuration.
This however does not mean that the algorithm will not have any knowledge about the processed models type and language as the metamodel also has to be known to the algorithm to be able to generate operation names for the created modelling delta. The metamodel must therefore also be provided to the difference calculation algorithm in a

representation that is applicable to the internal model representation.
Because model as well as metamodel representation are required to be as generic as no model-specific configuration is necessary, user-set difference calculation algorithm configurations should also be avoided if possible.
The resulting output should be as minimal as possible to reduce storage space costs of the modelling delta. This includes the avoidance of redundant information. Such information can occur when an element has been changed between two consecutive versions of a model but instead is being detected as a removal of the original and a creation of the changed element. Erroneous results, especially false-positive difference detections, must also be avoided by any means to reduce storage space costs.
The performance of the algorithm has yet only been spoken of in terms of storage space costs. As modelling projects of software systems might consist of large diagrams, consisting of hundreds of model elements per diagram [11], time efficiency also is a key requirement for the algorithm if it should be used in a productive system. Imagining the case that some changes have been applied to such a model and should be stored in an MVS, users should not have to wait for minutes until the difference information has been gathered. Whenever possible, runtime complexity optimising methods should be used by the algorithm to reduce computational time. This should however in no case cause worse difference detection correctness.
The problem of model difference calculation is similar to subgraph-isomorphism problem which is NP-complete but can be solved in polynomial time for certain graph types [19, p.202]. Assuming a simple model comparison where every element in the first version of a model is compared to every element in the second version, the resulting runtime complexity would be $O(n^2)$ where n is the number of model elements for both models. Concluding from these facts the main requirements for runtime complexity will be (1) finding a solution which is in the polynomial class and (2) that outperforms a runtime complexity of $O(n^2)$.
The complete list of requirements therefore is:

1. No user-set configuration is required.
2. Models can generically be processed only by using internal representations of their metamodel and the models itself.
3. The resulting output is a backward delta represented in the operation-based delta description language proposed by D. Kuryazov [16] as well as a creation delta in the same representation which generates the newer version of the model.
4. Variable assignments are consistently reassigned if elements had an assignment in a previous delta and new ones are assgined uniquely.
5. The generated deltas must be complete.
6. The generated deltas must be minimal.
7. The generated deltas must be correct.

8. The runtime complexity of the algorithm is in the polymial class.

9. The runtime complexity of the algorithm outperforms a runtime complexity of $O(n^2)$ where n is the number of elements for both processed models.

The following section will focus on an investigation of existing model difference calculation algorithms to find a general algorithmic solution that can be used to satisfy these requirements.

3. Related work

In this section several algorithms from recent scientific publications will be investigated to determine principles that could be used for an algorithm that could generally solve the model difference calculation problem. Each algorithm investigated will be described by listing its preliminaries, the algorithm itself with pseudocode implementations of useful methods, and an evaluation whether the algorithm could solve the targeted problem in general. Such a complete survey is necessary because the algorithms might significantly differ in performance (runtime complexity), difference detection correctness and difference minimisation.
In general, the difference calculation between two models is similar to the graph isomorphism problem which means finding the correspondences between two graphs. This problem is NP-complete [19] which is why a more specialised and outperforming algorithmic solution is needed. The algorithm requirements from section 2.6 should be satisfied apart from the ones focusing on output format which are specific to the targeted system boundaries from section 2.

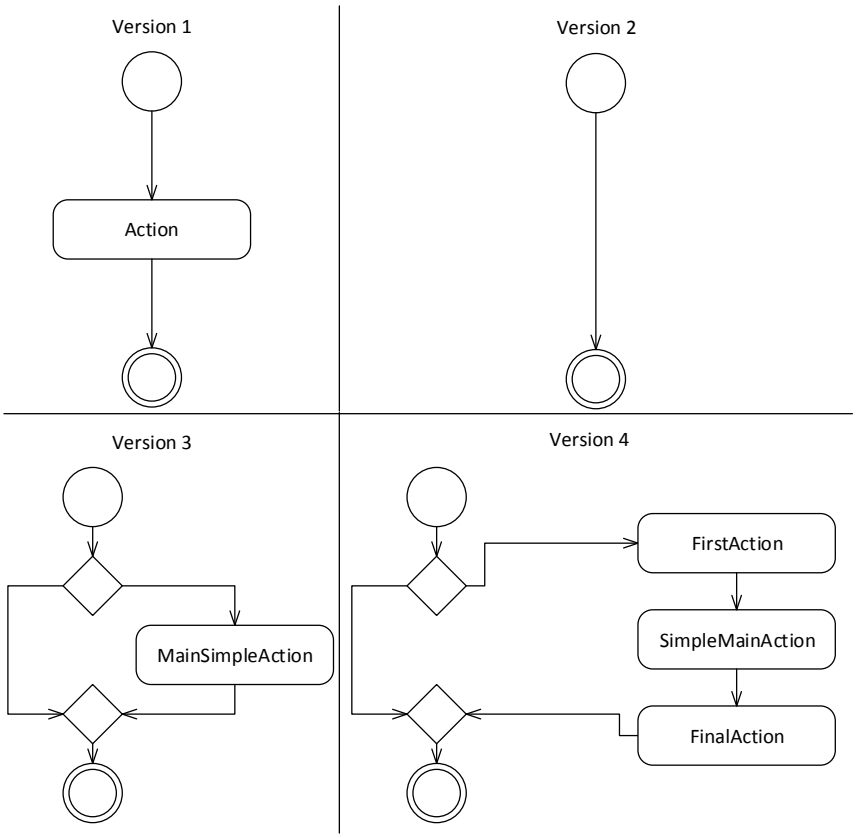

Figure 9: Four consecutive versions of an UML Activity Diagram

Whenever needed the example given in figure 9 is used to explain methods of the investigated algorithms. This example is a slightly modified and extended version of the example used in section 2 and represents four consecutive versions of the same UML Activity Diagram.
The abstraction level of the investigated algorithms will advance with the section, which means that algorithms that have been proposed to work only for specific model types (e.g. UML activity diagrams), will be investigated first and more general ones to the end of this section. The section however begins with an investigation of three specific approaches, giving details on persistent identifier based approaches, textual difference calculation, and the principle of semantic difference calculation and how usable they are given the context of this work. The rest of the section will focus on syntactic difference calculation methods which are most suitable for the targeted embedding.

3.1. Persistent-identifier-based approaches

Most internal model storage formats, for example the ones used in the UML modelling tools Visual Paradigm [9] and Rational Software Architect [1], are based on some proprietary XML representation of the models with each model element having a unique identifier assigned by the modelling tool. This also applies for models which are exported to XMI.
An obvious approach to model difference detection is given by these unique identifiers: By checking whether an element identifier occurs in both versions of the processed model it can be concluded whether that element has been changed, removed or added. Such an persistent identifier based approach was introduced by M. Alanen and I. Porres in 2003 [20].
The performance of such algorithms is only limited by the data structure representing the models because the only computation necessary is to check whether an element identifier is mapped to a model element. This can for example be done by using hash maps.
These approaches however have a major disadvantage: Due to the fact that model element identifiers must be persistent over various versions of the same model, it must be guaranteed that they never change whenever the model is changed. This might be true if only one modelling tool is used but as soon as the model is edited in more than one modelling tool, or even on more than one machine, this cannot be guaranteed. For example, repeated model exports to XMI from the Rational Software Architect [1], result in the same or different unique identifiers, depending on whether or not the user marks a specific checkbox.
Such tool-specific configurations cannot be taken care of by an MVS as well as a model difference calculation tool so it can be assumed that algorithms, based on persistent identifiers, are not a suitable solution for the embedding in a productive MVS.

3.2. Diff

One of the standard tools for comparing text files and calculating differences between them is Diff for Unix [21]. Text diff tools are part of every current source code manage-

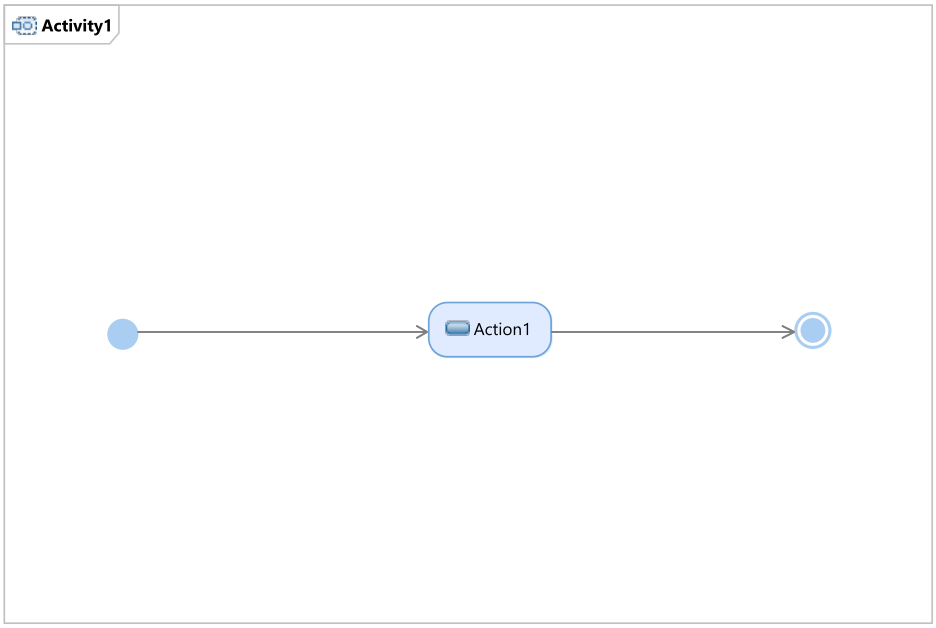

Figure 10: UML Activity Diagram created with Rational Software Architect [1]

ment system, especially used to handle conflicts between a working copy and the remotely stored files. Diff is capable of detecting character removals, additions and line changes between two compared files. This capability could also be used to detect differences between models if models would somehow be transformed into a textual representation. This is, for example, possible by exporting models into the XML Metadata Interchange (XMI) format [8]. XMI is a standardised format proposed by the OMG and commonly supported by modern UML modelling tools such as the Rational Software Architect [1] or Visual Paradigm [9]. The structure of the represented model is stored in XML form and can, due to the standardisation of the format, be exchanged between different tools. To explain the shortcomings of text based model comparison the activity diagram given in figure 10 has been created twice using the Rational Software Architect. Each of the creations resulted in the same model but elements were added in a different order meaning that in one version the Initial Node was placed first while in the other version it was placed last. An extract of the resulting XMI files to which the models have been exported is given in listing 1.
Although the given example is of the most simple structure, the differences between the XMI files are significant. The identifiers, which modelling tools assign to each element uniquely, are not identical. But as these are tool specific and should be ignored by a difference calculation tool, they can also be ignored in this case.
More important is the fact that the XMI files differ in the order of elements. If the models would be compared using a textual difference calculation approach, such a change of order would have been detected as either an removal and an addition or as a move of the

```
// XMI of the first version
<node xmi:type="uml:InitialNode"
        xmi:id="_6QjJC2kLEeK8-t9yg3WQbA"
        outgoing="_6QjJDmkLEeK8-t9yg3WQbA" />
<node xmi:type="uml:ActivityFinalNode"
        xmi:id="_6QjJDGkLEeK8-t9yg3WQbA"
        incoming="_6QjJEWkLEeK8-t9yg3WQbA" />
<node xmi:type="uml:OpaqueAction"
        xmi:id="_6QjJDWkLEeK8-t9yg3WQbA"
        name="Action1"
        outgoing="_6QjJEWkLEeK8-t9yg3WQbA"
        incoming="_6QjJDmkLEeK8-t9yg3WQbA" />

// XMI of the second version
<node xmi:type="uml:OpaqueAction"
        xmi:id="_td_nIWkMEeK8-t9yg3WQbA"
        name="Action1"
        outgoing="_td_nJ2kMEeK8-t9yg3WQbA"
        incoming="_td_nJGkMEeK8-t9yg3WQbA" />
<node xmi:type="uml:InitialNode"
        xmi:id="_td_nImkMEeK8-t9yg3WQbA"
        outgoing="_td_nJGkMEeK8-t9yg3WQbA" />
<node xmi:type="uml:ActivityFinalNode"
        xmi:id="_td_nI2kMEeK8-t9yg3WQbA"
        incoming="_td_nJ2kMEeK8-t9yg3WQbA" />
```

Listing 1: Extracts of XMI files created from the model given in figure 10

affected lines. The model elements that correspond to these lines of text however have not undergone any changes. The calculated difference is therefore incorrect because the information retrieved is redundant.
This type of incorrect change detection could be compensated by normalising the XMI files, respectively the XML structure of the files, so that such changes of order are somehow mapped to each other on a higher level. This would require another level of XMI preprocessing between the pure XMI file and the model comparison tool. This is certainly possible without too much effort but nonetheless it does not change the fact that Diff, as a standalone comparison tool, is not capable of correctly determining changes to models on XMI level.

3.3. Semantic difference calculation approaches

Most of the approaches investigated here are based on heuristics, such as model element names and structures, to determine differences between models on a syntactical level. But there are some approaches that try to solve difference calculation on a semantic level. Models that significantly differ in structure may indeed implicate the same abstracted view on a system.
For example there might be many ways to visualise the process of a shopping checkout with UML activity diagrams. If all of the traces, that lead from the initial node of the

model to the final node of the model via different activities or fork and merge nodes, exist in two models, those models are semantically the same. However they don't need to be of the same structure and differences between those models that are calculated on a syntax basis might not be empty.
Semantic difference detection finds differences that are not directly detectable on a syntactical level. Such differences can for example reveal bugs that have been fixed or features that have been added between two versions of the same model. These detection capabilities allow further tracing of model, and thereby system, evolution [22].

3.3.1. Approaches

One of the semantic approaches is the semantic difference calculation operator ADDiff proposed by S. Maoz, J.O. Ringert and B. Rumpe in 2007 [22]. ADDiff is capable to handle UML activity diagrams [14, pp. 319-434] and detect semantic differences between two given models of this type. Semantic differences of activity diagrams are described as *diff witnesses* which are execution traces that are possible in the initial activity diagram and that are not possible in the later one. Such an execution trace is a direct path starting from the initial node to the final node of the activity diagram via control flows, activity, fork and merge nodes. The checked activity diagrams are therefore translated into finite automata with input variables over finite domains. These automata are then checked using the *Symbolic Model Verifier* (SMV) [23] against specifications that allow the retrieval of such semantic differences. The operator aims on providing minimal difference sets by only checking traces, that are not prefixed by a trace that has already been classified as a difference trace. Additionally, it only returns one diff trace per possible assignment of the input variable values. These are adjusted over the finite range of possible values to detect all differences [22].
Another semantic diff operator has also been proposed by the same authors under the name of CDDiff [24]. CDDiff detects semantic differences between UML class diagrams [14]. Semantics of class diagrams are given in terms of object models with objects and the relationships between them. The operator checks for such object models that are possible in the first and not possible in the second class diagram. The operator outputs these differences again as a set of diff witnesses. These object models provide concrete proof about the difference in meaning of the compared class diagrams. But because such a set could be infinite when class diagrams are computed, the operator limits the semantic difference detection to a user-specified limit $k \in \mathbb{N}$. Thereby only object models where less or equal then k object instances exist are detected and added to the diff witness set [24]. CDDiff uses Alloy [25] to compute the detection. Alloy is a textual modelling language based on relational first-order logic. The compared class diagrams are transformed into an Alloy module with a transformation written by the authors. The module is then checked using Alloy to see if class diagram specific relations hold [24].
The concrete process of detecting the semantic differences will not be deeply investigated in this work as semantic differencing does not solve the problem of this work as explained next. As both of the briefly described algorithms were proposed by the same authors

they share most of their methods of comparing models. But some of these methods can be considered common for semantic approaches in general.
The foundation of semantic differentiation forms the definition of semantics for processed model types. In the algorithms described, these are given by the semantics of UML class and activity diagrams. To detect semantic changes, both algorithms use a use case specific helper tool that performs logical checks on the models according to the processed semantics and returns semantic differences in a way that is specific to the processed model type. These helper tools are specifically chosen and configured by the authors, to be usable for such a processing.

3.3.2. Evaluation

Semantic approaches are not of use for the model versioning systems, targeted in this work. While it is a valuable feature to detect semantic changes between successive model versions and thereby conclude more tracing information, it is significantly harder if not impossible to deterministically use information from a semantic difference and apply it to a model to generate another version of the same model. While such a set only contains informations about features or bugs that are not present within one version of a model, it does not contain the information that is needed to remove or add this feature to a model to obtain the other version. This would have to be done in a special step where model difference information is somehow transformed into a set of syntactical difference operations. If these syntactical differences are detected directly and then globally used in the model versioning system embedding, no such transformation is necessary. Furthermore, because models that differ in their structure can have the same meaning, models that have the same meaning might have a totally different structure. It is therefore not clear if such a set of syntactic difference operations can be deterministically concluded from the semantic information. Additionally the described algorithms are not generalisable to additional model types as both are based on underlying systems, SMV and Alloy, that include special semantic and logical calculation methods specific to the handled model type.
Semantic difference detection is still a feature that could improve the functionality of a model versioning system by informations that are not even included in current source code versioning systems but in the context of this work it is apparently of no further use.

3.4. UMLDiff

UMLDiff has been proposed by Z. Xing and E. Strouli in 2005 [12] and aims on providing a difference calculation algorithm to obtain information about structural changes between two versions of a software system. The information obtained enhances change informations that are detected by using certain source-code metrics or clone detection methods and is capable of detecting refactoring changes such as moving features along classes, restructuring of data structures and classes, and changes to the interactions between components.

Despite the fact that the algorithm's main use case does not match the goal of this thesis, it can still be viewed as a model difference calculation algorithm and contains several ideas that can be used to calculate model differences in general. The algorithm will therefore be described in detail here although some of the descriptions will only be of use for understanding the principles behind but not for a later evaluation of the algorithm.

3.4.1. Preliminaries

UMLDiff takes two Class Diagrams that are reverse-engineered from two versions of an existing, object-oriented software system (source code) as input and outputs the structural differences as a so-called *change tree*. The software-system is reverse-engineered into class diagrams by reading the source code files of the system, which are itself residing in a source-code versioning system, and transforming it into class diagrams [12] that conform to a metamodel that is specifically designed for this algorithm. Although UMLDiff is specifically designed to handle UML Class Diagrams, the used principles are adapted to UML Activity Diagrams for the course of this section.
The generated model spans a containment tree above the language specific programming constructs of the software system. The programming language used by the authors is Java which is why the following containment elements are named the same way as constructs from the Java programming language are called. The root of the containment tree is referred to as the *Virtual Root* which represents the system as a whole and that can contain *Packages*. Packages contain top-level *Classes* and *Interfaces* which declare fields, methods, inner classes and interfaces. Classes additionally declare fields, methods, constructors and initialisers. These declarations are also interpreted as containment. Fields may contain an initialiser and blocks contain local and anonymous classes which means that blocks can be contained by methods and field initialisers. Relations between these elements conform to the principle of UML dependencies [14] and can be of type containment, declaration, inheritance, interface implementation, field read or write, method call, class creation, field data type, method return type, parameter type and exceptions that are declared or thrown.
The same containment order can be applied to UML Activity Diagrams for example when viewing a subset of Activity Diagram elements containing only activites, control flows, initial, final, action, decision and merge nodes. Activities are in this case the root of the diagram containing any other type possible, including other activites.
The containment tree consists of nodes and edges where all the containment elements (package, class, interface, field, block) build nodes and the relations between those elements build the edges. The system can then be traversed top-down by the algorithm advancing from the virtual root of the system, which also is the root of the containment tree, to the leaves which are blocks or fields. In case of Activity Diagrams it would therefore traverse from activities down to contained activites, nodes and control flows.
Such a tree is built for both compared versions of the software system and then compared by the algorithm to detect structural differences. Within the original approach the created trees are stored in a relational database in PostGreSQL and the structure of

the graph is represented by the structure of the database tables. The graph traversal is being performed by database queries and pre-defined views on the database [12].

3.4.2. Algorithm

The algorithm computes structural differences from two input models that are generated from the source codes according to the metamodel described before. The system is, by transforming it to the described tree, represented as a directed graph. The containment order of the elements in the model define a logical partial order which in the case of UML Activity Diagrams would be

$$\begin{aligned}\text{Activity} >(&\text{Initial Node, Final Node, Action Node,}\\ &\text{Decision Node, Merge Node, Control Flow)}\end{aligned}$$

which allows a top-down traversal on the graph beginning at the root [12]. UMLDiff traverses the trees of both created models simultaneously and tries to identify elements that correspond in the way that an element exists in both version of the model. Important to note is the fact that UMLDiff compares elements logical level by logical level. That means that activities from both versions are compared first and when the set of activities has been completed, the algorithm advances to the comparison of nodes and flows, starting with comparing one type of nodes. This reduces the amount of elements to be compared and makes processing the tree multiple times unnecessary. The comparison is done in two steps by checking (1) for the same or a similar name and (2) for similar relations to other elements that already have been marked as existing in both versions of the system [12]. Both of these steps use special heuristics which are described in detail.

Name Similarity Unlike other difference calculation algorithm such as DSMDiff, this algorithms does not solely try to match equally named elements but instead also checks if similarly named elements could be a candidate for the currently checked host element. This heuristic is referred to as *Name Similarity* and allows the detection of renamed elements. The name similarity metric used in UMLDiff is based on the longest common substring (LCS) algorithm that is commonly used to compare strings and which returns a value between 0 and 1 depending on the lexical similarity between two strings. But LCS does not return acceptable results when the compared strings are similar in the way that only their order of words is different. Therefore a new metric is used in UMLDiff that is based on the amount of common adjacent characters in two compared strings s_1, s_2:

$$\frac{2 \cdot |\,\text{adjacentCharacters}(s_1) \cap \text{adjacentCharacters}(s_2)|}{|\,\text{adjacentCharacters}(s_1)| + |\,\text{adjacentCharacters}(s_2)|} \tag{1}$$

The function $adjacentCharacters(s)$ returns a set of character pairs that occur as adjacent characters in the string s. A pseudocode implementation of the name similarity algorithm is given in listing 2.

```
double nameSimilarity(s1,s2)

// Get every adjacent pair of characters from both compared strings
1. HashSet pairs1 = pairs(s1.toUpperCase());
2. HashSet pairs2 = pairs(s2.toUpperCase());
3. int union = pairs1.size() + pairs2.size();
// Keep only the pairs that occur in both strings in pairs1
4. pairs1.retainAll(pairs2);
5. int intersection = pairs1.size();
6. return intersection * 2.0 / union;
```

Listing 2: Pseudocode implementation of the name similarity algorithm [12]

The string comparison can be used on version 3 and 4 of the example given in figure 9 where multiple action nodes in version 4 are candidates for the single action node in version 3. Obviously to the human eye is the fact that the action node `MainSimpleAction` has been renamed to `SimpleMainAction` and that these elements are therefore corresponding. Using LCS on all three action node names in version 4 would result in all of them having the same longest common subsequence to `MainSimpleAction`. However, the correct choice `SimpleMainAction` is least similar because the shared subsequence is shorter in relation to the length of the complete string. The adjacent character pairs algorithm of UMLDiff would correctly detect `SimpleMainAction` as corresponding because it contains the highest amount of shared adjacent character pairs.

Structural Similarity Every element is additionally checked using the structural similarity metric. In the original approach the relations between entities are based on the containment order described before. A pseudocode implementation of structural similarity computation is given in listing 3.
When an element is being structurally compared to a candidate element, two sets are being created, one for the host and one for the candidate element, containing all of the elements that the element is connected to, via a specific relation type. These sets are then intersected using a special equals function that checks whether names of the contained elements are similar, by using the name similarity metric, or whether some of the elements have already been matched.
The method `getCount(element,element,relation_type)` returns how many times the given elements are connected via the given relation type. This is used to count the amount of relations that are connected to matched elements as well as the amount of relations that are connected to unmatched elements.
Using these values, the formula given on line 19 returns a higher similarity the more matched elements are connected and a smaller similarity the more unmatched elements are connected to the checked element. If the checked element is not connected to any other element via the given relation type, the name similarity is used with an exponent that increases with the amount of checked relation types. The effect of the name similarity is therefore dampened for elements with very few to none connections. As structural similarity is computed per relation type, the structural similarity results are normalised by the amount of possible relation types for the processed model type [12].

```
double structureSimilarity(e1,e2,relation_type)

 1. Set e_of_r1 = getEntitiesOfRelation(e1,relation_type);
 2. Set e_of_r2 = getEntitiesOfRelation(e2,relation_type);
 3. if (e_of_r1.size == 0 and e_of_r2.size == 0)
 4.     pow++; return power(nameSimilarity,pow);
 5. int beforecount = 0, aftercount = 0;
 6. for all er1 in e_of_r1 and all er2 in e_of_r2
 7.     if (er1.equals(er2)) {
 8.         beforecount += getCount(e1,er1,relation_type);
 9.         aftercount += getCount(e2,er2,relation_type);
10.         e_of_r1.remove(er1);
11.         e_of_r2.remove(er2); }
12. int beforecount = 0, afterleftcount = 0;
13. for all er1 left in e_of_r1
14.     beforeleftcount += getCount(e1,er1,relation_type);
15. for all er2 left in e_of_r2
16.     afterleftcount += getCount(e2,er2,relation_type);
17. int min = min(beforecount,aftercount);
18. int max = max(beforecount, aftercount);
19. return min*1.0/(max+beforeleftcount+afterleftcount);
```

Listing 3: Pseudocode implementation of the structural similarity algorithm [12]

```
double computeSimilarityMetric(e1,e2)

1. nameSimilarity = nameSimilarity(e1.name,e2.name);
2. int pow = 0;
3. double metric = 0.0;
4. for all relation_type in possible relation types
5.     metric += structureSimilarity(e1,e2,relation_type);
6. int N = amount of possible relation types;
7. return (nameSimilarity + metric)/(nameSimilarity+N);
```

Listing 4: Pseudocode implementation of similarity computation [12]

Similarity Composition Both of the similarity metrics described before are used in conjunction for every element as seen in listing 4. Unlike other approaches, UMLDiff does not solely match elements on the fact that one metric, for example the name similarity, returns equality but also takes structural similarity into account for every element. This conjunction of similarities is specifically used to allow the detection of element moves and renamings. Those detections require separate user set thresholds that must be reached to deem moved or renamed elements as equal and their value directly affects detection correctness. Whenever more than one element exceeds one of the thresholds, the one with the highest similarity is chosen.
The found difference facts collected by UMLDiff are unchanged, renamed and moved elements. Additionally, from elements that have not been matched in the first version and the elements that have not been matched in the second version of the diagram,

additions and removals can be concluded [12].

3.4.3. Evaluation

UMLDiff has been specifically designed to obtain software evolution information from an existing software system that has been transformed into a model. UMLDiff therefore does not directly cover the target of this work but the mechanisms described in detail are certainly of use for a more general, model-focused use case.
UMLDiff is missing a runtime complexity estimation of the algorithm but instead there are some absolutely measured test cases given by the authors [12]. Estimating the runtime complexity of UMLDiff it can be assumed that it is $O(n^2)$ in a worst case scenario where every element is of the same type. In all other cases the top-down hierarchical type-by-type traversal of the models should significantly reduce the required element comparisons.
The similarity mechanisms of UMLDiff to detect renamings and compare elements by structural similarity can certainly be of use when handling any model type. Especially the name similarity metric, which returns better results than LCS on mostly unchanged identifiers (word order), seems useful.

3.5. SiDiff

SiDiff is a framework for model comparison that has been actively developed at the University of Siegen since 2004 and that claims to be totally metamodel independent as long as the processed instance of a model can somehow be represented in a graph-like structure [2]. SiDiff exists in a variety of development branches focused on very different modelling domains. One branch provides model comparison on MATLAB/Simulink diagrams which are commonly used by engineers in the areas of signal and image processing, another one adapts SiDiff for a toolbox that is used in automotive development, and finally there has been some recent effort on using SiDiff to detect similarities between sentences of a constraint language or to compare molecular graphs [2].

3.5.1. Preliminaries

SiDiff processes two input models that are given in a textual or binary format which is either complying to XMI [8] or a proprietary format. SiDiff provides a parser that transforms these models into an internal representation of the models. The internal representation of the models is a typed, attributed and directed graph whose metamodel is given in figure 11. Every node corresponds to an element in the original model and edges represent existing connections between those elements. Every node in the graph is additionally connected to a type that represents its elements type and may have a set of attributes consisting of pairs of keys and values. Edges likewise are connected to a type representing the connection in the model and have a boolean attribute whether the connection is a reference or a nesting edge in case of part-of semantics of the model [2].
To reduce the computational cost when trying to find corresponding elements from two versions of a model, a special high-dimensional search tree, the S3V (Similarity search

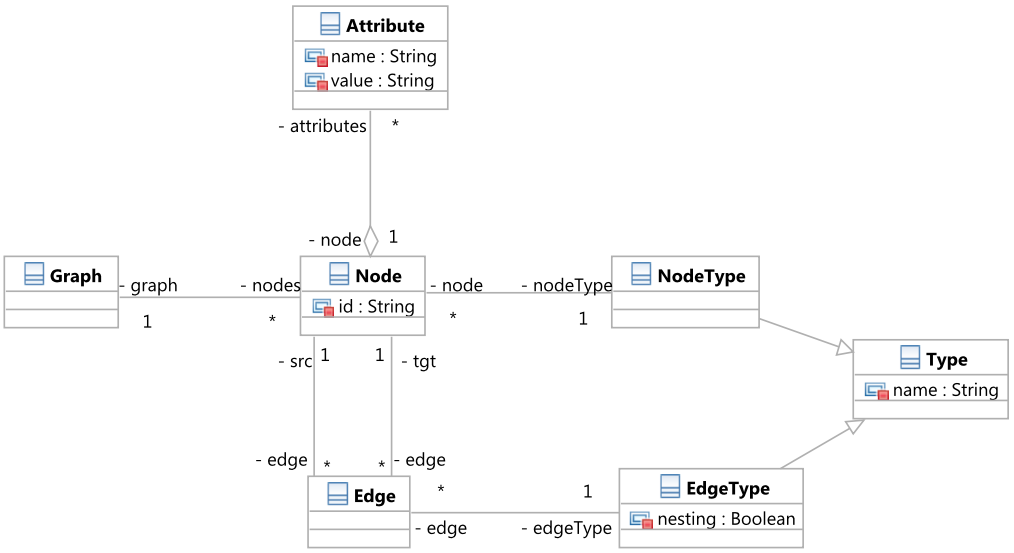

Figure 11: SiDiff internal graph metamodel[2]

Criterion	Weight
Similar value for attribute *name*	0.5
Equal value for attribute *state*	0.1
Similar elements following outgoing *Edge type A*	0.1
Similar elements following incoming *Edge type B*	0.1
Matched parent element	0.2

Table 1: Examplary SiDiff similarity configuration [2]

sparse vector) tree, is used to store the model elements in memory. To find possible matching candidates from another model usually all of the model elements must be checked for similarity. Instead of having to compare all elements the elements are stored in the S3V tree according to their similarity to each other. Every element is therefore interpreted as a numerical vector where every index represents a certain characteristic of the element. The similarity of two elements is then defined as the euclidean distance with smaller distances resulting in higher similarity. The tree can then be used to find a set of most similar elements of a given element by doing a range query on the tree which returns all similar elements within a subspace of the tree defined by the specified range or distance of the query.
One S3V tree is created for each of the compared models and for each occuring element type before beginning the comparison. All of the trees are completely kept in memory for the time of the comparison [11].

3.5.2. Algorithm

Like most of the difference calculation algorithms the SiDiff algorithm searches for elements that occur in both compared models first. This is done by calculating similarities between the elements and marking the most similar elements from both models as corresponding. The similarity between two elements is given as a float value between 0 (no similarity) and 1 (equality). Similarity is usually determined by local attributes or elements in the near proximity of the investigated element. SiDiff provides a set of standard functions for such similarity measurements but also allows the addition of new similarity functions when needed. A basic similarity functions would for example be given by assigning a similarity of 1 if the names of two elements are equal and add this similarity with a weight of 0.5 to the totally calculated similarity. Some more examples are listed in table 1 [2]. For string similarity comparison the longest common subsequence (LCS) algorithm is used [26].
The exact similarity calculation on two given models is specified by providing a configuration file to SiDiff. The comparison can consist of a set of similarity functions as similarity might not in all cases depend only on one similarity factor. The similarity functions specified can additionally be weighted to mark certain functions more important than others and giving them a greater effect on the similarity between two elements which is overall computed as the weighted mean over the result of all considered similarity functions. The minimum similarity needed to mark two elements as corresponding can also be defined within the configuration file and is referred to as the *similarity threshold* [2].
The matching approach is based on the concept of similarity flooding where the similarity of two nodes increases when nodes in their neighbourhood are similar. That means that whenever a node has been matched, the matching has to be propagated to all nodes in the near proximity. The algorithm therefore processes the graphs in a bottom-up/top-down manner [2].
The algorithm begins a bottom-up traversal of the graph by checking all nodes of an element type for possible matches in the candidate model. In the original approach, where UML class and object diagrams are processed, models are assumed to specify a hierarchical order of element types. It is thereby possible to start at a leave type of this order and process the type order bottom-up. As not every model's metamodel allows the conclusion of such an order it is assumed that the algorithm processes the element types in a model independent order, for example alphabetically [26]. For UML Activity Diagrams it can be assumed that such an order would be given by handling containers, activites or partitions, first and contained element afterwards.
Nodes are identified as similar, if the similarity threshold is reached. However, nodes are only matched, if they are uniquely similar which means that only one candidate node exceeds the threshold. If more than one node exceeds the threshold no matching is performed immediately, instead the node will be reinvestigated after nodes in the near proximity have been matched [2].
As soon as a node has been matched the bottom-up processing of all the nodes of a given type is interrupted. The second phase, a top-down propagation of the newly detected

match, starts. The algorithm checks all nodes that are directly connected to the previously matched node for possible similarities using the same similarity functions as used in the bottom-up phase before. If a new match is found during this step, the similarity propagation is again invoked for the elements neighbouring the new match. When all neighboured nodes have been investigated, the neighbourhood investigation stops and the algorithm returns to the bottom-up phase and continues processing the nodes of a specific type [26].
Given the example in figure 9, the concept of switching to the top-down processing would for example occur after the decision nodes in version 3 and 4 have been identified as a match. Instead of further investigation of decision nodes or the next type that is to be processed, all elements that are connected to the matched decision node would be investigated. In that specific case all incoming and outgoing control flows would be investigated for possible matches. If one of them is matched, they will not have to be investigated when there type is processed and they immediately lead to another neighbourhood investigation.
The algorithm stops when all nodes and types have been investigated during the bottom-up phase [26]. However it is possible to let the algorithm start a new iteration over all the node types and nodes to detect new similarities that might have been caused by matches found when checking a node type that is processed at the end of the bottom-up phase. Because it cannot be assumed that the processed graphs are of pure tree structure and therefore might contain cycles which could cause the algorithm to process the models infinitely, the algorithm stops after an iteration that did not notice any new matches [2]. This is specifically needed when the processed models consist of less composite structures but rather of graph-like references [26].
Some optimisations could be applied to SiDiff to achieve a better runtime when searching for correspondences. To identify possible matching candidates before the start of the similarity computation and iteration over the graph, elements from both models could be hashed by using a hash function that returns a hash value based on the local attributes of the element. When two elements from the models share the same hash value they can be matched without having to process them in the manner of similarity flooding. Another method to enhance the matching by pre-process functions is given by using fingerprints. Fingerprints are bit strings representing the characteristics of an element. The technique has been used in the context of search queries in molecular databases. The calculation of fingerprints allows a quick guess if similarity calculation is worth the computation but the approach is also limited as element types need to have enough characteristics to produce a usable fingerprint [2].
Whilst only corresponding matched elements have been identified yet, changes to the elements, additions and removals still have to be detected and stored for later use. This is done within the so-called unified document which contains the relevant information of the difference calculation. The unified document contains all elements of both compared models and the difference information. Already matched elements are however only listed once in the document. SiDiff classifies differences into four categories:

- Structural differences: Elements that have been added or deleted

- Attribute differences: Elements that are corresponding but that have changed attributes
- Reference differences: Elements that are corresponding but that have changed references
- Move differences: Elements that are corresponding but that have a changed parent element

The unified document thereby is a representation of a symmetric difference [2].

3.5.3. Evaluation

SiDiff is the most generally usable framework to calculate model differences surveyed in this work. The many already developed use case specific branches underline the flexibility of SiDiff. The active development, enhancing the original framework from 2004 with hash functions, fingerprints and the usage of S3V trees make SiDiff seem like an elaborated product that has gone through several development cycles.
The runtime complexity of SiDiff is pessimistically estimated to be $O(n^2)$ on models with only one element type where n is the number of elements. This places the SiDiff algorithm in the polynomial complexity class and thereby outperforms the graph isomorphism problem [2]. The listed enhancements which are partly implemented, specifically hash functions, fingerprints and S3V trees, further increase the performance in terms of runtime complexity as less similarity comparisons of elements are needed. This seems necessary as the authors state that medium to large documents consisting of a few hundred elements, lead to runtimes of 5 to 60 minutes [11]. Imagining a productive model versioning system where differences have to be computed for every changed model whenever updates are to be committed, this is certainly unsatisfying. The introduction of S3V trees alone however improved the runtime by a factor of up to 50 when processing large models [11]. This leads to a more usable time effort although several minutes are still quite an amount of time when working in a productive system. The introduction of hashing functions further reduces the runtime complexity to $O(n \log n)$ and thereby also significantly speeds up the matching process [11].
The SiDiff algorithm detects, additionally to the three basic difference categories required for the embedding described in the previous section, moves of elements within models which is an enhancement of the difference information. Due to the fact that SiDiff only matches elements of the same type, the detection of type changes is impossible. SiDiff also provides similarity computation flexibility in the way of providing a way to define a set of similarity functions which allows customisation of the calculation process on a more detailed level. However, it is necessary to define such a set for every processed model type separately according to the available elements. The transformation of the processed models into a directed graph conforming to a model type independent metamodel allows the single algorithmic approach on which SiDiff is based. Thereby any model type can be processed without having to modify the algorithm whilst the transformation of the processed models into the internal graph structure must still be

somehow defined. The similarity configuration and the required configuration file in general in conjunction with the model transformation make SiDiff quite user dependent. If the framework should be used in a productive model versioning system these settings have to be provided by the system for all processable model types without requiring user interaction. As any model type should be processable this is certainly difficult to achieve.

3.6. DSMDiff

DSMDiff (Domain Specific Model Differentiation) has been proposed by Y. Lin, J. Gray and F. Jouault in 2007 [13] and introduces a differentiation tool for domain-specific models. Compared to the previously investigated algorithms, that focused on difference calculation for any model type, this work tries to provide a solution for models that have been designed in a non-UML but domain-specific modelling language. Therefore, the algorithm used in DSMDiff should be capable of handling multiple metamodels for different domain-specific languages (DSL). Furthermore, it is claimed that the implemented algorithm is metamodel independent.

3.6.1. Preliminaries

To achieve metamodel independency the DSMDiff algorithm assumes DSLs to be defined along the Generic Model Environment (GME), a meta-configurable tool that allows DSLs to be defined from a metamodel. By using the GME approach, model elements are either *atoms*, *models* or *connections*. An atom is the most basic entity with no further internal structure whilst a model may contain other models or atoms. A connection represents a relationship between two entities. The model is thereby internally treated as a hierarchical tree which can be traversed from root to leaves.
Any of these elements consists of *type, kind, name* and a set of *attributes*. The type describes whether it is a atom, model or connection. The kind represents the name of the defining element in the metamodel which would be for example final node when speaking of UML Activity Diagrams. The name is the identifying descriptor of a types instance in a model. Connections are extended by source and target attributes known as *source* and *destination* [13].
With this structure any domain-specific model becomes a hierarchical graph where models are nodes, atoms are leafs, and connections are the edges between nodes and leafs. With this tree-like structure the differentiation algorithm can traverse top-down, starting from the top-most model root and traversing down the tree to all the atom-leafs at the end.

3.6.2. Algorithm

A pseudocode representation of the algorithm is given in listing 5. The DSMDiff difference calculation algorithm basically consists of two steps:

1. Element mapping

```
Name: DSMDiff
Input: diffModel
Output: diffModel

1. Initialize a set hostSet and a map candMap;
2. Get the host model from diffModel as M1 and the
   candidate model as M2;
3. Detect attribute differences between M1 and M2 and
   add then to the Change set of diffModel;
4. // Find node mappings by signature matching
   findSignatureMappingsAndDeleteDiffs(diffModel,
                                       hostSet,candMap);
5. If (hostSet is not empty && candMap is not empty)
        // Find node mappings by structural matching
        For each element e1 in hostSet
                1) Get its candidates from candMap into a set
                   called candSet;
                2) e2 = findMaximalEdgeSimilarity(e1,candSet);
                3) Add Pair(e1,e2) to the mappings set of diffModel;
                4) Erase e1 from hostSet;
                5) Erase e2 from candMap;
6. If (candMap is not empty)
        Add all the remained members of candMap to the new set of
        diffModel;
7. For each mapped elements that are not submodels
        Detect attribute differences and add them to the Change
        set of diffModel;
8. Compute edge mappings and differences
9. // Walk into the child submodels
   For each childDiffModel that stores a pair mapped submodels
        DSMDiff(childDiffModel);
```

Listing 5: Pseudocode implementation of DSMDiff [13]

2. Difference detection

Before checking for differences between two processed models `m1` and `m2`, the algorithm tries to find elements that exist in both models. These elements are then stored in a set named the *Mapping Set*. The mapping set contains these mapped elements as maps as in `Map(elem1,elem2)`, which means that the element `elem1`, which exists in the model `m1`, also exists in the model `m2` as the element `elem2`. The algorithm checks model nodes, atom nodes and connections separately and in this order for potential mappings as seen in listing 5 [13]. Regarding the exemplary Activity Diagram given in figure 9, the initial, action, final, decision and merge nodes would be represented as atom nodes whilst the control flows would be represented as connections. The activities surrounding the nodes, which are omitted from the figure, would be a model node and the same would apply for all other elements available in Activity Diagrams that somehow form a composite structure, for example partitions or action nodes with in- and outgoing pins.

```
Name: findSignatureMappingsAndDeleteDiffs
Input: diffModel
Output: hostSet, candMap, diffModel

1. Initialize a set hostSet and a map candMap;
2. Get M1 from diffModel and store all nodes of M1 in hostSet;
3. Get M2 from diffModel and store all nodes of M2 in candMap
   associated with their signature;
4. For each node e1 in hostSet
        1) Get the count of the nodes from candMap that are
           signature matched to e1;
        2) If count == 1
                Get the candidate map from candMap as e2;
                Add Map(e1,e2) to the mapping set of diffModel;
                Erase e1 from hostSet;
                Erase e2 from candMap;
        3) If count == 0
                Add e1 to the Delete set of diffModel;
                Erase e1 from hostSet;
        4) if count > 1
                Do nothing;
```

Listing 6: Pseudocode implementation of signature mapping [13]

Signature Matching To find elements that can be mapped the algorithm checks every node or edge in the models by performing a *signature check* which is given in listing 6. The signature of an element is defined as the concatenation of its type, kind and name, or when speaking of connections as the concatenation of the name of its source, its type, kind, name and the name of its destination. An element occurs in both models if `Signature(elem1) = Signature(elem2)` which reduces the matching to a simple string comparison. However it might be possible that more than one node or edge is found in the target model `m2` that has a matching signature. The source node in `m1` and all the candidate nodes in `m2` are then postponed for another matching method [13].

Structural Matching When trying to structurally match possible target nodes from `m2` to the source node in `m1`, the algorithm checks incoming and outgoing edges of the nodes for so called *edge similarity* as in listing 7. The edge similarity metric between two nodes is formally defined as the number of edges with equal signatures connected to a host node in `m1` and a candidate node in `m2`. The algorithm then selects the candidate node with the maximum edge similarity as the unique mapping for the host node. If such a node cannot be found because some candidate nodes have equal edge similarities, the algorithm chooses one of the candidate nodes as the mapping because it can be assumed that model elements may occur multiple times in a model [13].
After the mapping is completed, all unchanged nodes are contained within the mapping set and the algorithm can now proceed to the determination of model differences. DSM-Diff considers model differences to be categorised into additions, removals and changes

```
Name: findMaximalEdgeSimilarity
Input: hostNode, candidateNodes
Output: maximalCandidate

1. Initialize maps: hostConns, candConns and set
   maxSimilarity = 0, maximalCandidate = null;
2. Store each edge signature and the number of associated
   edges of the hostNode in the map hostConns;
3. For each candidate c in candidateNodes
        1) Store each of its edge signatures and the number of
           associated edges in the map candConns;
        2) Call computeEdgeSimilarity(hostConns, candConns) to
           compute the edge similarity of c to hostNode;
        3) if (the computed similarity > maxSimilarity)
                maxSimilarity = the computed similarity;
                maximalCandidate = c;
4. Return maximalCandidate;

Name: computeEdgeSimilarity
Input: hostConns, candConns
Output: similarity

1. Initialize the similarity as zero;
2. For each edge signature in the map hostConns
        1) Get the number of the edges associated with the
           edge singature as hostCount;
        2) Get the number of edges from the map candConns
           associated with the edge signature as candCount;
        3) If candCount <= hostCount
                similarity += candCount;
        4) Else
                similarity += hostCount
3. Return similarity
```

Listing 7: Pseudocode implementation of edge similarity computation [13]

of model elements. To detect removals, the algorithm performs a signature check to find elements that exist in `m1` but not in `m2`. This is done within the mapping step of the algorithm by including a special step that marks elements as deleted if neither signature nor structural matching could find a unique candidate element for a given host element. These elements are added to the delete difference set. After that all the elements from `m1` should be contained either in the mapping set or the delete difference set. The elements from `m2` that are not contained in the mapping set have not been mapped to a host node in `m1` and have therefore been added and are now added to the new difference set. All the nodes in source and target model should now be contained within one of the three created sets but the signature and structural mappings ignored the attribute sets of the nodes completely. Thus, the attributes of mapped nodes are now checked for differences by comparing the values of each attribute of a pair of mapped nodes. The changed values are then added to a fourth set, the change difference set [13].

Because until now only nodes of the models have been investigated, edges are still to be done. The edge difference calculation consists of four steps that partly conform to the previous matching techniques and should need no further explanation:

1. Every edge that is connected to a node that is contained within the delete difference set is being added to the delete difference set.
2. Every edge that is connected to a node that is contained within the new difference set is being added to the new difference set.
3. Edges are being mapped from source to target model by performing signature matching.
4. Unmapped edges in `m1` are added to the delete difference set; unmapped edges in `m2` are added to the new difference set. [13]

3.6.3. Evaluation

To have a foundation for evaluating this algorithm a investigation of the runtime complexity seems necessary. To estimate the runtime complexity of this algorithm several assumptions are made.

1. Compared models are similar in structure in size so that node and edge count are approximately the same
2. The number of edges is significantly smaller then the number of nodes and edge mapping can thereby be omitted from the runtime complexity estimation.
3. Signature matching and edge similarity computation require the most computational work and other steps are thereby omitted from the runtime complexity estimation.

The worst case scenario for this algorithm is a complete graph where all nodes are directly connected to each other via edges and where no node mapping pair can be found within the signature matching step. For signature matching every host node has to be checked for a unique mappable candidate node which makes the upper bound for this computation step $O(N \cdot \log(N))$ where N is the number of nodes in (both) models. Because no direct map can be found by using signature matching, edge similarity has to be computed for every host node and the computational cost for this step is bound by $O(T \cdot N \cdot R \cdot (N-1) \cdot \log(N-1))$ where T is the total number of nodes summarized over all levels of the multi-level model and R is the number of candidate nodes with $R \leq N$. The runtime complexity of the algorithm is thereby concluded to be in the polynomial class [13] and therefore outperforms the general graph isomorphism problem if all the previously given optimistic assumptions apply.
The algorithm however has a major disadvantage when submodels have been moved from one place to another between two checked models. The algorithm is unable to detect such changes and instead computes a removal from the old position and an addition

to the new position of the complete submodel into the model tree. An improvement is proposed to add another difference operation *move* to the set of available operations [13]. But as the difference operations of this work's embedding is limited to additions, removals and changes this fact is ignored for a possible implementation of this algorithm. The algorithm assumes models to be given in a multi-level structure that is being processed top-down recursing into the submodels. This is not an assumption that has to be made for any model type but it certainly makes the algorithm more usable in general. The process given together with its runtime complexity and the detection correctness could provide us with a solution that could be used to calculate differences on any model type.

3.7. Summarisation

The investigation of algorithms has been focused on finding an algorithm that is generally capable of handling model difference calculation for any modelling language and model type. Apparently, all of the investigated tools and frameworks fail to provide this generality but they provide a general algorithmic approach.
Apart from algorithms that do detect semantic instead of syntactic differences and the ones that require model elements to have persistent identifiers, the remaining serious model difference calculation algorithms are unsuitable due to the following facts:
Due to the original purpose of UMLDiff [12], which is the detection of differences between the source codes of software systems, it is incapable of handling any model type other than its internal model representation. Its methods of model traversal, name and structural similarity however are useful for a generic handling of models.
The SiDiff [2, 11] framework is the first investigated framework that provides a difference calculation algorithm that is as generic as desired. SiDiff however needs model type specific user-set configuration files without which no model can be processed. The concept of similarity flooding as well as the internal representation of graphs are however suitable solutions to certain problems when handling models generically.
DSMDiff [13] at last, uses methods similar to SiDiff and UMLDiff but its signature and structural matching methods seem limited in comparison to the metrics used by UMLDiff because of its incapability to, for example, detect moved elements. Additionally, DSMDiff expects models to conform to the Generic Model Environment. This requires that the metamodel for each model to be processed, has to be somehow transformed into the Generic Model Environment format. On the other hand, DSMDiff just like UMLDiff uses a top-down hierarchical order of model elements on which the models are traversed showing that this is somehow a convenient approach. Additionally, DSMDiff provides detail on how to track differences on implementation level using its so-called difference sets.
Although this investigation did not provide a solution that is as generic as desired, the general algorithmic approach to model difference detection is the same for all investigated tools and frameworks. Thereby, the investigation provides a solution to the first problem of this thesis, a general algorithm for model difference calculation which will be generalised in the following section.

4. GDiff - A General Algorithm for Model Difference Calculation

The algorithms and frameworks for model difference calculation investigated in the previous section, provided a general algorithmic solution to model difference calculation which is one of the main goals of this thesis. The algorithm is given in figure 12.

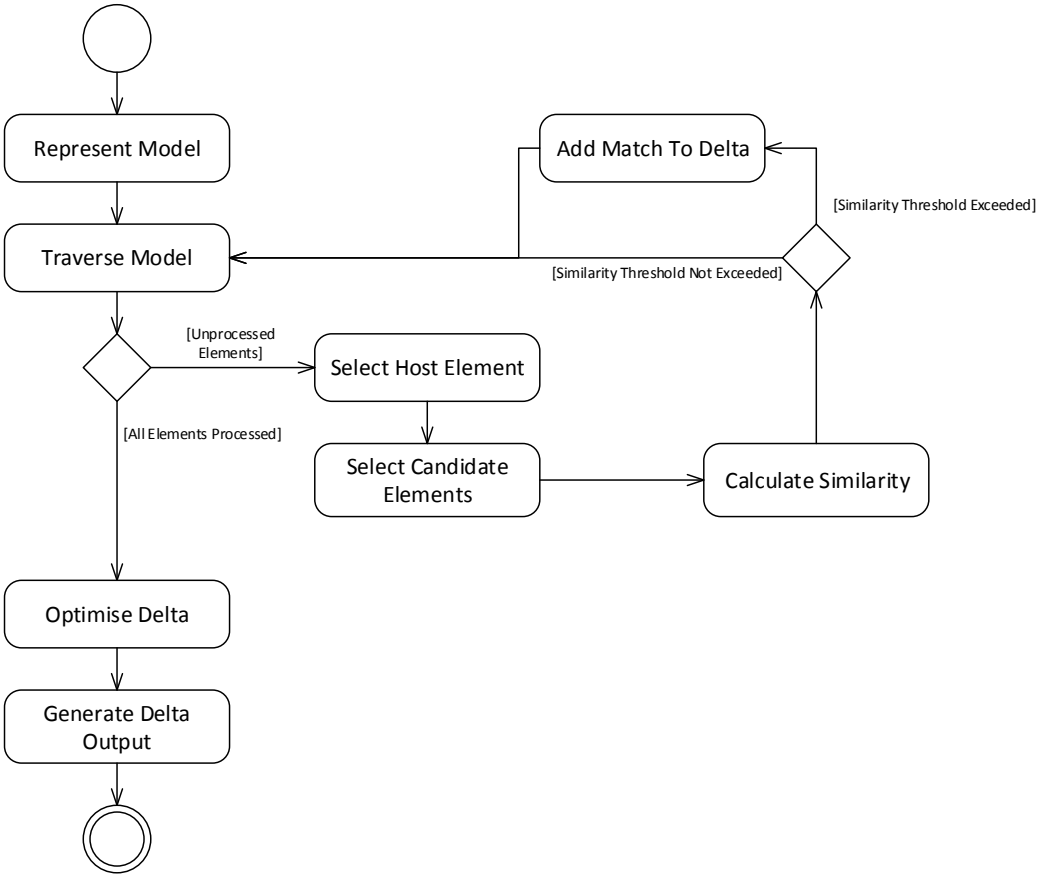

Figure 12: General algorithm for model difference calculation

The figure shows the basic steps in a model difference calculation with omitted object flows. The process of model difference calculation however has to be provided with two consecutive versions of the same model, referred to as host and target model. In the case of the MVS that builds the system environment, as explained in section 2.5, the host model would be the newer, the target model the older version of the same model due to the fact that backward deltas are generated.
The actions in figure 12 occur in every previously investigated algorithm. First, an in-

ternal *model representation* is needed to make the models generically processable. The models are then *traversed*, for example in a top-down hierarchical order, to investigate every element and search for possible matches between the two compared models. If there are still elements to be compared while traversing the model, one of them is *selected as host element* and *candidate elements*, possible matches, are gathered from the target model. The similarity between the host and all candidate elements is then *calculated using similarity metrics*. If at least one of the candidate elements similarity exceeds the internal similarity threshold, the candidate element with highest similarity is *selected as a unique match* for the host element. After all elements have been processed the *delta output is generated* in a specific delta representation.
One additional step has been added to this figure, the *delta optimisation* right before the delta output generation which takes care of output specific delta preparation caused by the used delta representation which is generated in the delta output generation activity. This component could however be used for further, not output specific optimisations which are not considered during this thesis.
After the first problem of this thesis, the search for a general algorithm, has been solved, the algorithm now has to be implemented in a way that satisfies the required generality as well as all the requirements from section 2.6. The implementation, using methods from the previously investigated tools and frameworks, is described in this section, structured according to the activities of the model difference calculation algorithm given in figure 12. The goal is to have a standalone prototype called *GDiff* (Generic Model Difference Calculation) that is capable of handling any model type and language at the end.

4.1. Architecture

The prototype implementation of GDiff has been conducted in the Java programming language making use of its object-oriented nature. Actions from figure 12 are represented by several components. GDiff itself is then simply the composition of these components as seen in figure 13, implementing the control flow. For example it retrieves the next host element to be investigated from the model traversal component and passes it to the similarity computation. After similarity computation, `GDiff` handles the result by either adding the found match to the delta or continuing to the next host element. After all elements have been processed, `GDiff` initialises the delta optimisation and output generation. In the specific context of GMoVerS it takes care of the backward as well as the creation delta generation.
The implementation is kept expandable by generalising every component by an interface. This way each component can later be exchanged for a possibly more enhanced version. Table 2 shows the mapping of components to actions from figure 12 showing that some components provide multiple actions. The action "Model Representation" is missing a component in figure 13 as well as a mapping in table 2 due to the fact that it is taken care of by a third-party library.
Some of the components might depend on each other. The similarity computation component, for example, is relying on the model traversal component because the traversal component provides methods to obtain elements of of a specific type from a model.

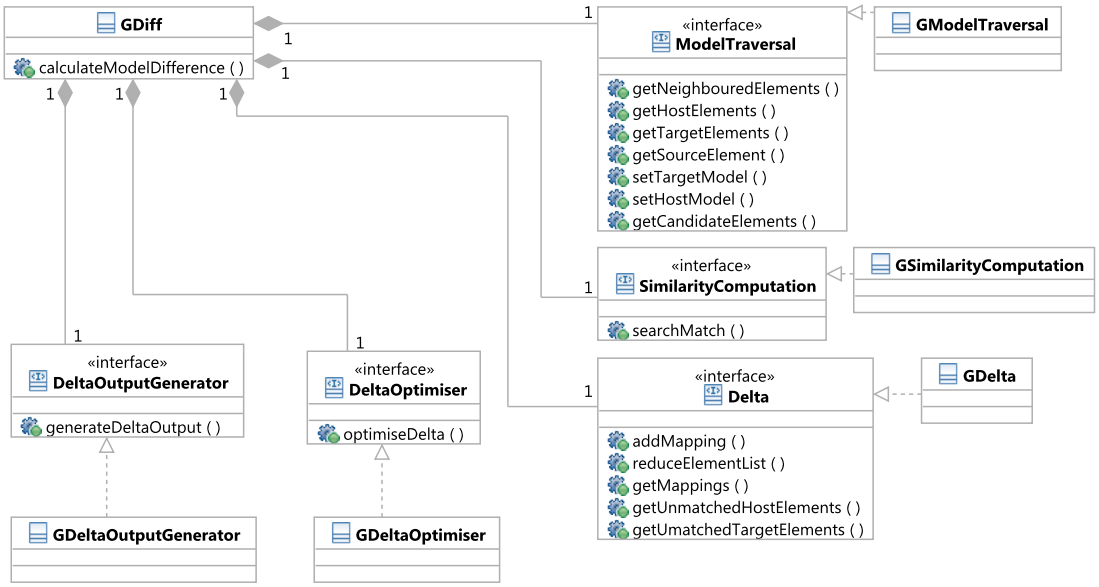

Figure 13: Composite structure of the GDiff implementation

Component	Provided Actions
ModelTraversal	Traverse Model Select Host Element
SimilarityComputation	Select Candidate Elements Calculate Similarity
Delta	Add Match To Delta
DeltaOptimiser	Optimise Delta
DeltaOutputGenerator	Generate Delta Output

Table 2: Mapping of GDiff components to algorithm actions

Due to the desired expandability of the prototype, such dependency shall not be restricted and therefore a simplified form of dependency injection is used. One component, in the case of GDiff it is the component which aggregates all other components, implements the interface `ComponentProvider` and allows access to all components used by the implementation. This `ComponentProvider` must be provided to every `DifferenceCalculationComponent`, the base class for every component, whenever one is instantiated so that it can retrieve its component dependencies from the `ComponentProvider`.

4.2. Model Representation

As stated before when determining the requirements for a general approach for model difference calculation, a general format, on which the algorithm can process models inde-

pendently from their original language and type, is needed. Every investigated algorithm also uses an implementation-specific format for this purpose.
The graph-like approach, although needing user interaction in the case of SiDiff [2, 11], seems most suitable for the required generality. The best choice seems to be defining a metamodel, similar to the one used by SiDiff, which allows the representation of any model as a graph, which contains information about the structure of the metamodel of the processed model.
GDiff uses TGraphs [27] to represent models. A TGraph is a general graph class which conforms to the fundamental mathematical construct of graphs consisting of vertices and edges. Addtionally, TGraphs are typed, attributed, ordered and directed. Hence, every vertex or edge conforms to a type and can have attributes. Edges outgoing a vertex additionally are directed and ordered which means that they have a defined source and target vertex and can be iterated in a defined order.

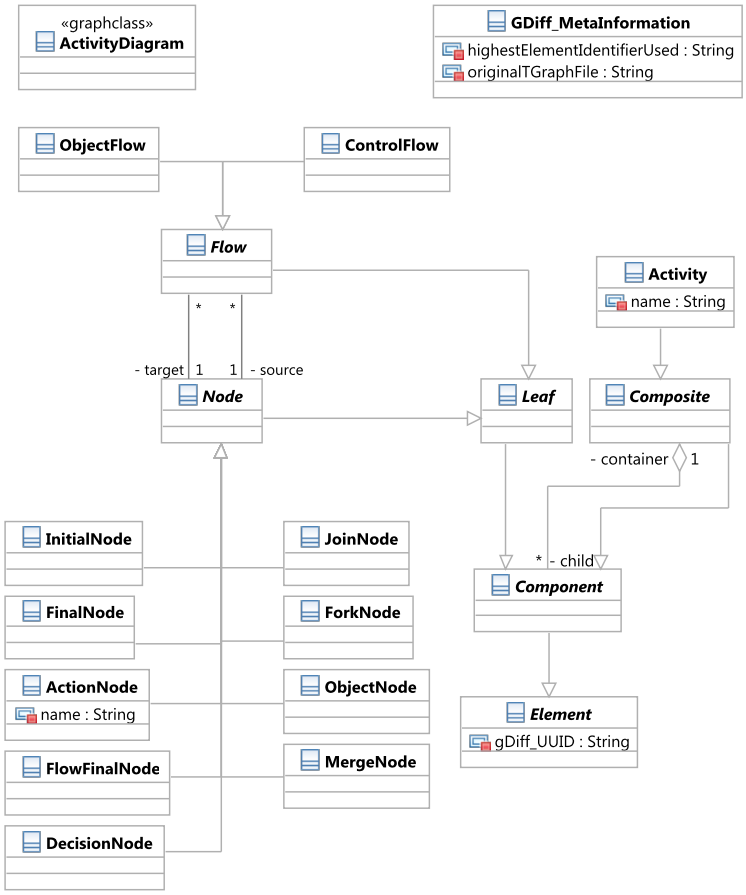

Figure 14: TGraph schema for a subset of UML Activity Diagrams

It is easily possible to represent any model as a TGraph by defining a schema for it and transforming it to the TGraph representation. Such a schema is given in figure 14

for the previously introduced subset of UML Activity Diagrams. As a matter of fact, this schema equals the simplified UML Activity Diagram metamodel given in section 2, figure 2. Every class in the UML Class Diagram representation of the schema results in a vertex in the TGraph representation. Thereby, all nodes and Activities as well as flows will result in vertices after transforming such an Activity Diagram to a TGraph. Abstract types in the given schema can not be instantiated in a TGraph and are only added to allow inheritance.

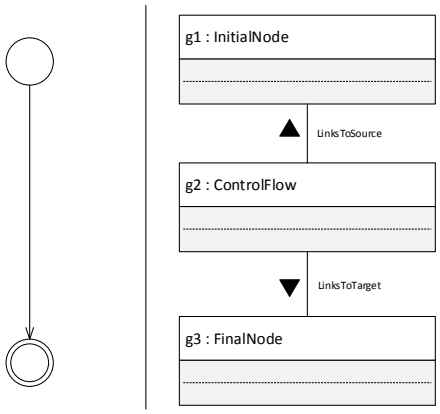

Figure 15: UML Activity Diagram (left) and the corresponding TGraph representation (right)

Figure 15 shows an Activity Diagrams representation as a TGraph using UML Object Diagrams where each class instance represents a TGraph vertex. The object identifiers `g1`, `g2` and `g3` have only been added for model consistency and are not part of the TGraph. Each element in the Activity Diagram on the left is directly represented by a vertex on the right. Each of these vertices is of a certain type which is given by the class of the object which directly maps to the type of model elements in the Activity Diagram. Hence, the types of represented model elements can directly be retrieved just by retrieving the type of a TGraph vertex.
Additionally, the TGraph metamodel in figure 14 makes relationships and associations between model elements also representable by vertices. The Control Flow in the Activity Diagram in figure 15 is represented by a vertex of type `ControlFlow`. This vertex is connected to the source and target vertices of the Control Flow by edges which also conform to a specific edge type resulting from the metamodel of the TGraph.
By using this approach only vertices have to be considered for model difference calculation on TGraph level. Every versionable element of the model is indeed represented by a vertex whilst all the edges solely representing the structure of model elements. They are not to appear in the generated delta.
The schema however contains three non abstract types that contain metainformation either for TGraph processing or model difference calculation. The `ActivityDiagram` element of stereotype `<<graphclass>>` is required for TGraphs and is only used for in-

ternal schema identification. The other two elements, `GDiff_MetaInformation` and the abstract supertype `Element`, have been added to satisfy the requirements from section 2.6. They are used to add information about previously created modelling deltas of previous versions of the processed model. The attribute `gDiff_UUID` can contain delta variable values that have been assigned to an element in a previous delta. The metainformation element contains the value of the last used delta variable identifier so that new delta variables can be uniquely assigned.
Due to the generic graph structure of TGraphs any model is generally transformable into a TGraph representation. This requires an additional preprocessing step before the model difference calculation implementation GDiff which transforms models from their original representation into the TGraph format.
A tool for the transformation of UML Activity Diagrams has been provided by the GMoVerS project [15]. The tool transforms models that were created with the modelling tool Rational Software Architect [1] and exported to their XMI representation into TGraphs.
Due to the necessity of this preprocessing, the input and output parameters are changed in contrast to the parameters given in section 2.6. Additionally, the delta metainformation previously introduced result in a toolchain for model difference calculation as given in figure 16.
Instead of GDiff receiving two models directly as input, they are preprocessed by an external activity used to transform the models into their TGraph representation. This activity also requires the metamodel of the processed models so that an according schema can be generated. The schema is implicitly contained within the generated TGraph files.
The metainformation required for GDiff is added to the target model TGraph by another utility, the GUUID Applier which has been implemented specifically for this thesis. It has to be provided with the TGraph of the host model and the creation delta for that model which has previously been generated. It adds delta variable values to all TGraph vertices by setting the corresponding attribute in the TGraph. Additionally it correctly sets the attributes of the metainformation element.
The resulting TGraph outputs of transformer and applier are then used as the model input for GDiff which can thereby also directly access the schema of the TGraphs and has access to previously assigned delta variables and element identifiers. The two preprocessing steps additionally clean up the signature of GDiff as it now only receives two TGraphs as input rather than receiving two TGraphs, an element identifier and a creation delta. To generate creation deltas for the first version of a new model, GDiff is also capable of running in single model mode which results only in a creation delta for the given model.
To handle TGraphs on implementation level the third party library *Java Graph Laboratory* (JGraLab) [28] is used. This library provides functionalities to handle TGraphs in general to, for example, load TGraphs from the file system or to inspect their schema, contained vertices and edges. It basically provides any graph traversal and investigation functionalities required for the purpose of this work.
The library is used by every component described in this section meaning that they directly access the internal TGraph representation on implementation level using JGraLab.

JGraLab is the only third-party library used within GDiff and also the only restriction to model handling implied by the architecture.

4.3. Delta

While performing the model difference calculation on TGraphs, some delta representation on implementation level is necessary to be able to keep track of elements that have already been matched between host and target model. The `Delta` component is providing capabilities for this purpose.
The generic functionality of the component is to provide access to all matched and unmatched elements from both models as well as methods to check whether a specific element has already been matched or to add a new match.
The actual implementation of the interface `GDelta` serves its purpose by making use of Java collections. The implementation works according to the new, delete and change difference sets used by DSMDiff [13].
The component is initialised with all elements from host and target model which are then kept in internal lists. For matched elements, an instance of the Java collection `Map` is used to map host model elements to matched target model elements. The map is directed in the manner of host model elements being the key, returning the matched target model element if the key is requested. As soon as two elements have been matched to each other, they are removed from the initial host and target model elements list. Thereby, all elements are only contained in one collection of the delta, either in the host or target element list, representing unmatched elements, or the map, representing already matched elements.
All of these collections can be retrieved by using standard getter and setter methods. Additionally, the component provides a boolean check whether an element has already been matched without needing to access the map directly. This works for host as well as target model elements. Therefore, it can easily be checked if an element has already been matched, for example, before it is selected as a candidate element.
This internal delta representation can also be used to generate the delta output in a later step as given in section 4.7 as the lists and map directly represent added, deleted and changed elements.

4.4. Model Traversal

Every element from the compared host and target model must be investigated for a possible match in the opposite model. To guarantee this, the models have to be orderly traversed meaning that each element is at least investigated once.
For this purpose, the component defined by the interface `ModelTraversal` is responsible. The interface declares a method which shall return the next element to be investigated while comparing the model. That element can then be investigated by other components to find possible matches. An implementation therefore must keep track of previously investigated elements and define an order by which elements are returned. Preferably, this is done in a runtime optimised way so that the amount of necessary element compar-

isons is reduced. The element retrieval can however be restricted by providing a boolean parameter to the method, telling whether or not the last returned element resulted in a new match. This can be used to adjust the model traversal method whenever new matches occur.
Model traversal in GDiff is restricted to a single model: The declared method always returns elements from the host model, making the search for matches unidirectional. This does however not falsify the resulting difference.
Additionally, the component serves as a generic model provider, providing methods to retrieve host and target model completely or certain elements by type. Therefore, the traversal component can be used by other components to gather, for example, candidate elements for similarity computation.
The model traversal in GDiff returns host elements to be investigated in the manner of all surveyed algorithms by generating a top-down hierarchical order of element types. This hierarchy is based on composition of elements, adding containers on top and contained types below their containers.
Elements are thereby investigated by type level starting with the top most type. In the case of the given subset of UML Activity Diagrams, the top most type is given by Activities, being the type that can contain any other type, even itself. When starting the comparison of two Activity Diagrams, the component `GModelTraversal` would return an Activity as the first element to be investigated, advancing to one of its contained types after all Activities have been investigated.
This top-down traversal by type is however interrupted whenever the previously investigated element resulted in a new match between host and target model. According to the investigation of SiDiff [2, 11], the concept of similarity flooding significantly reduces the amount of required element comparisons. Hence, similarity flooding has been implemented in the component `GModelTraversal`.
The flag telling whether or not the last returned element resulted in a match is used for this purpose. If that flag is `true`, the type traversal is interrupted and all elements that are directly connected to the previously returned element are gathered. The next host elements will then be returned from this neighbourhood until all neighboured elements have been processed. A neighbourhood investigation can again be interrupted if a new match occured, resulting in a recursive neighbourhood investigation starting from that new match.

4.5. Similarity Computation and Element Matching

Elements selected for investigation, have to be checked against candidate elements from the target model to find corresponding elements. The interface `SimilarityComputation` has been defined for this purpose, providing a method that tries to find a matching element for a given element from the host model. Because candidate elements are not provided to the method, the similarity computation component can and must select a set of candidate elements from the target model itself for example by gathering elements of a specific type from the `ModelTraversal` component.
The calculation of similarity as well as the selection of the most similar element is taken

care of by the actual implementation. In the case of GDiff this implementation has been conducted in the component `GSimilarityComputation`. The implementation relies on several similarity algorithms that are implemented using the strategy pattern [29], allowing simplified exchange and extension of similarity algorithms.
These similarity algorithms accept two model elements and return a floating value representing the similarity of the two elements. The value is in $[0, 1]$ with 0 meaning no similarity and 1 meaning equality.
The prototype implementation of GDiff uses two specific similarity algorithms provided by surveyed model difference calculation tools.

4.5.1. Name Similarity

For model elements that contain a name attribute, such as Activities or Action Nodes when speaking of UML Activity Diagrams, name similarity is computed using the algorithm described and used in UMLDiff [12]. That algorithm is based on adjacent character pairs and seems adequate for the model element similarity purpose.
However, because it cannot be generally assumed that a named element in any model always has a unique attribute going by the identifier "name", the attribute which contains a possibly unique name is generically detected by GDiff respectively the similarity algorithm. This is done by simply checking for attributes for which the attributes name contains the substring "name". Whenever no such attribute can be found for the compared elements, the resulting name similarity is 1 because similarity should not be reduced only due to the fact that unnamed elements are being compared.

4.5.2. Structural Similarity

For all model elements, structural similarity is computed using the method described and used in UMLDiff [12]. This similarity algorithms considers any incoming and outgoing edge of the compared elements, checking whether already matched elements are connected to them. This results in the similarity algorithm requiring access to the current state of the delta between the models. Thus, similarity algorithms are implementations of the interface `DifferenceCalculationComponent` themselves and have access to any component used within GDiff.
In the original description of structural similarity, neighbourhoods of the compared elements were being checked for elements with similar names and already existing matches. This resulted in name similarity of neighboured elements being used for relation types that do not appear for the investigated elements. As name similarity is already being used as a standalone similarity algorithm in GDiff, being computed for every element comparison, only already matched elements are considered for structural similarity. Hence, the partial structural similarity for relation types that do not appear for the investigated elements is 1. This also results in a structural similarity of 1 for compared elements that are not connected to any other elements.

4.5.3. Similarity Composition

Similarity algorithms used in the component are used in sequence. Unlike the similarity composition used in UMLDiff [12], where name similarity influences the computed structural similarity by invoking one algorithm from the other, GDiff invokes similarity algorithms in sequence and normalises the accumulated result to a similarity between 0 and 1. This is similar to the similarity configuration used within SiDiff [2], only differing in the fact that similarity algorithms are equally weighted in GDiff.

4.5.4. Match Selection

After the similarities have been calculated for each candidate element, one element might be deemed as a new match for the host element. Just like the surveyed algorithms, GDiff uses an internal similarity threshold which must be exceeded to deem two elements as corresponding. If it is not exceeded, then no match is added to the delta.
The exact, most optimal value for similarity threshold cannot be selected in general. For tests using the subset of UML Activity Diagrams as given before, a minimum required similarity of 0.5 seems adequate. This value is selected due to the fact that two TGraph vertices that represent the same unnamed element, for example an Initial Node, have a structural similarity of 0 if neither their container or incoming and outgoing Control Flows have been matched. Because an Initial Node is an unnamed element, the similarity threshold is still exceeded because the name similarity algorithm returns a similarity of 1. Hence, unnamed elements can be matched although there neighbourhoods might not contain any matches yet.
However, independent from a possibly changed optimal threshold value, as soon as the threshold has been exceeded, the host element and the candidate element with highest similarity are added to the Delta. If their is more than one element with highest, threshold exceeding similarity, then, in the case of the GDiff implementation, the first most-similar element that has been compared is added as a match.

4.6. Delta Optimisation

The component `DeltaOptimiser`, transposed by the implementation `GDeltaOptimiser` in GDiff, preprocesses the internal delta representation for an ensuing delta output generation in the specific delta representation given in section 2. The requirements 5, 6 and 7 in section 2.6 listed delta minimalism, completeness and correctness which might also be taken care of in a delta optimisation component. It can however be assumed that the algorithm result, specifically the detection of changed, unchanged, added and removed elements is taken care of by the model traversal and similarity metrics described before. The internal delta thereby already represents a correct and complete delta. The minimalism of the delta can only be affected by modifying the process of difference detection.
The purpose of this GDiff specific delta optimisation is simply to reorder the elements contained in the delta list so that the generated delta will be consistent no matter how far it is processed.

To explain the necessity of such an optimisation, the following listing contains an extract of a delta which changes the target of a Control Flow to a new Action Node. Delta variable `g1` is assumed to be defined outside of the extract:

```
g1.changeControlFlowTarget(g2);
g2 = addActionNode("Action");
```

This example highlights some of the problems of unordered delta outputs. The goal is to have a consistent delta in each line of the delta representation, meaning that the delta does not refer to not yet existing elements in a specific line. This problem already occurs in line 1 of the given delta because it changes the target of an Control Flow to an element variable that has not yet been assigned in the delta. This actually happens not before the second line of the delta.
To have a consistent delta, the order these two exemplary delta operations would have to be switched. This is the purpose of the component `GDeltaOptimiser` which rearranges the elements in the lists for unmatched host and target model elements contained in the `Delta` component. The rearrangement guarantees a specific order of unmatched target elements which represent added elements:

1. Elements that are added to a containing element are placed after the container in the list.
2. Items that are connected to another element are placed after the connected element in the list.

The inverse order is generated for elements in the unmatched host element list which represent deletions:

1. Elements that are added to a containing element are placed before the container in the list.
2. Items that are connected to another element are placed before the connected element in the list.

Requiring that change operations are added after add and delete operations in the delta and that add and delete operations are generated in the same order as added or removed elements are contained in the delta lists, this order guarantees a consistent delta.

4.7. Deriving the Modelling Delta from Calculated Differences

At last, the modelling delta has to be generated in the specific delta description language introduced in section 2.4. This job is taken care of by the delta output generation component, defined by the interface `DeltaOutputGenerator`. The actual implementation which is capable of generating the required delta description language is conducted in the component `GDeltaOutputGenerator`.
All elements have previously been investigated, trying to match elements from host and target model. The internal delta representation however does not directly reflect change information. It only contains lists of unmatched elements from host and target model.

The first directly represent removed elements, the latter added elements. Add and delete operations can therefore be generated by adding add or remove operations for each element in the lists.
Changed elements are however only contained in the map of the delta which holds all elements that have been marked as corresponding between host and target version of the model. Whether they actually changed or not has not been determined yet. Due to the internal model representation as given in section 4.2, each element from the processed model is represented by a vertex. This has a major impact on the process of delta output generation. The delta description language requires contained elements to be added on their container as well as requiring association elements to set their source and target elements as parameters within a add operation. Hence, contained elements as well as association elements have to be somehow detected generically.
Containments are directly represented by TGraph edges that are of a specific aggregation kind, either shared or composite. Whether the currently processed element is contained by another element can thereby be decided by checking all incoming and outgoing edges of the vertex for such an aggregation kind. If a corresponding edge occurs, the required delta operation can be generated without further investigation if it affects an add operation. To detect whether the containment changed in case of a matched element, the container of the matched host as well as the matched target element have to be retrieved. If these elements are also marked as matching in the internal delta representation, no change operation is necessary. Otherwise, a change operation is generated and added to the delta output.
Association elements can be uniquely detected by checking whether an element has exactly two outgoing edges which are not of aggregation kind shared or composite. A Control Flow from the metamodel given in section 4.2 is for example always connected to its source element via an edge of type `LinksToSource` and to its target element via an edge of type `LinksToTarget`. The type names of these edges are automatically generated when transforming the metamodel into a TGraph schema and it can therefore not be guaranteed that such naming conventions occur for every processed model type. If source and target element associations in the metamodel are however restricted to be identified by the names `source` and `target`, it can be assumed that the resulting edge types also contain these identifiers.
Thus, by checking whether an element has exactly two outgoing edges excluding shared or composite aggregation kinds, it is definitely an association. Furthermore, by checking which of the edges type names contains the keyword `source` or `target`, edges connecting the association to its source or target element can be identified. Generated add operation for newly added associations can thereby be created. For matched elements, it has to be checked whether the source or target of the host element are also matched to the source or target of the matched target element. If not, a change operation can be generated which changes the source or the target attribute of the association.
The variable identifiers used in the delta description language can be retrieved from the GDiff-specific metainformation attribute contained in the TGraph schema. For matched elements, these are propagated from target to host model before generating the delta. Removed elements which do not have an identifier yet, receive a new one by using the

highest element identifier used before, retrievable from the metainformation vertex, and increasing it with every unassigned element.
The last used identifier is then added to metainformation lines in the delta so that it can be reapplied to TGraphs in the GUUID Applier.

This generic implementation of a model difference calculation algorithm has been successively conducted. Whether or not this implementation actually works generically has still to be evaluated in the following section.

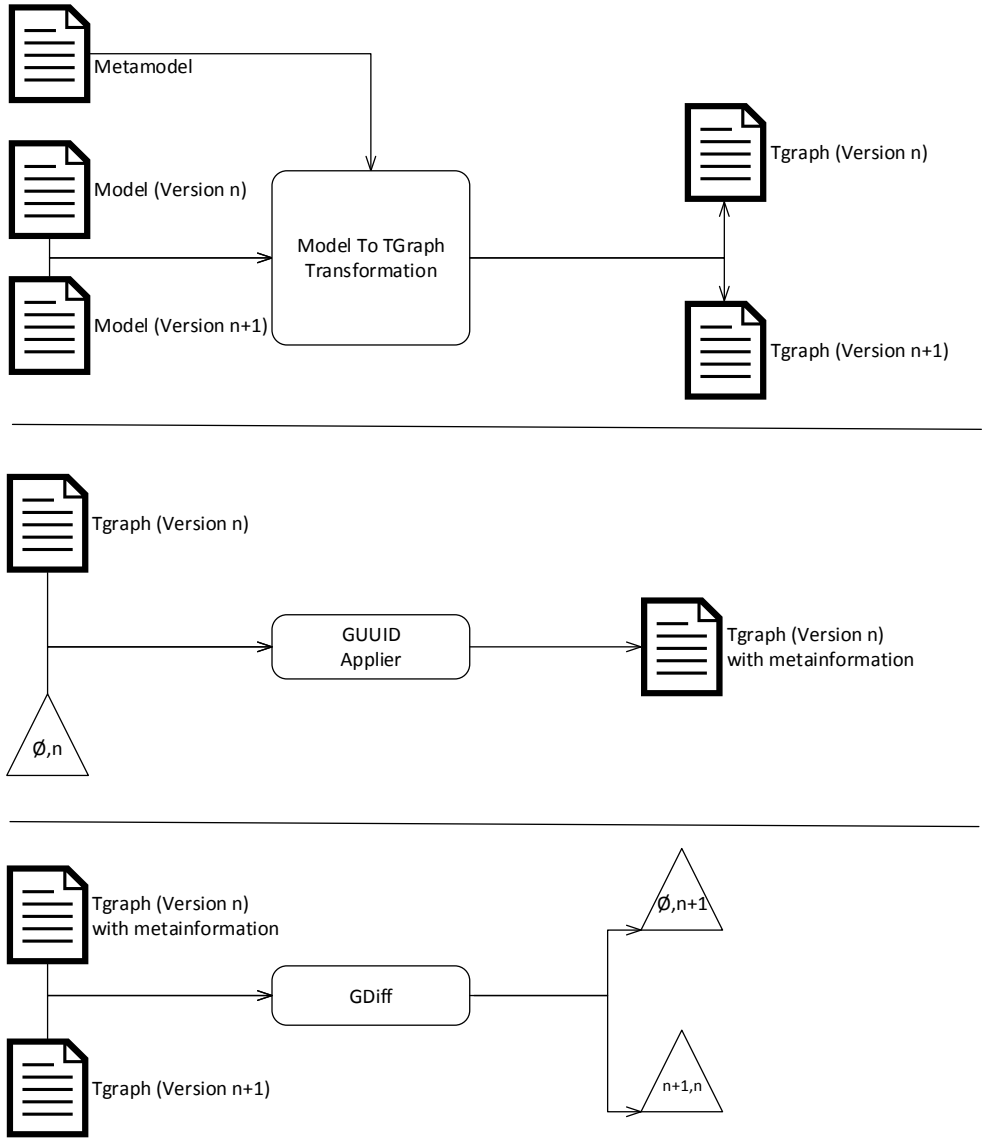

Figure 16: Model to TGraph transformation (top), Metainformation application to TGraph (middle), input and output parameters for GDiff (bottom)

5. Evaluation

This section aims on an investigation whether or not the implementation described in the previous section fulfils the requirements from section 2.6. This section is split into two parts according to the following evaluation criteria:

1. Evaluation of GDiff against UML Activity Diagrams
2. Evaluation of GDiff against TGraphs in general

5.1. UML Activity Diagrams

For the course of this work, UML Activity Diagrams have mainly been used as an example to model difference calculation and the principles behind. Additionally, a metamodel as well as a TGraph schema have been designed for a specific subset of UML Activity Diagrams. This can be used to test the implementation.
In section 2, three not empty, consecutive versions of an exemplary UML Activity Diagram have been introduced. Furthermore, an additional fourth consecutive version has been added in section 3. Each of those versions has been extended with an Activity going by the name "Activity" that contains all other elements. The diagrams have been created using Rational Sofware Architect, exported to XMI and transformed to TGraphs using the provided parser for Activity Diagrams. Thereby, this evaluation also serves as an evaluation for the GMoVerS-specific model difference calculation toolchain introduced in section 4.
The models are thereby comparable using GDiff. This is additionally possible without requiring any model specific configuration of GDiff, fulfilling requirement (1) and (2).
The expected result for versions 1 to 3 have previously been given in section 2 when investigating forward and backward delta calculation. When processing these 3 version in consecutive order, GDiff generates exactly those expected deltas, differing only in variable assignments and the order of operations. Delta variable assignments are also consistent over all consecutive versions, for instance the activity is assigned to the variable `g0` in all generated creation and backward deltas. Add, delete and change operations are generated for exactly those elements that they are expected for. Additionally, when processing version 3 and 4 of the model, the resulting backward delta again totally conforms to the expectations.
Thus, GDiff actually provides a correct delta, referring to the correct element types and generating correct delta operations even for contained elements and associations, fulfilling requirements (3) and (4). It can also be assumed that the chosen similarity threshold as well as the similarity algorithm composition work as desired on the exemplary diagrams. Hence, the resulting deltas fulfil the requirements of a (5) complete, (6) minimal and (7) correct delta.
Regarding runtime complexity of the algorithm and its implementation, the tests showed that runtime complexity $O(n^2)$ has indeed been outperformed due to the by type comparison and similarity flooding. Requirements (8) and (9) thereby are also fulfilled.

This small test case already proves that all requirements are fulfilled for this model-specific test case. Whether or not this applies for models in general may only be decidable when investigating another model type in the following section.

5.2. Java TGraphs for Software Evolution Investigation

The original purpose of GDiff is to calculate model differences that can be used in a model versioning system. To investigate how generically usable the implemented approach actually is, GDiff has been tested on TGraphs that represent a software system. The SOAMIG project [30] extends software migration to service oriented architectures. The repository of the projects approach is based on TGraphs to represent migratable software systems and artefacts [31]. The project offers a webservice based on a Java fact extractor [32] which transforms a software system given as Java source code into a TGraph representation.

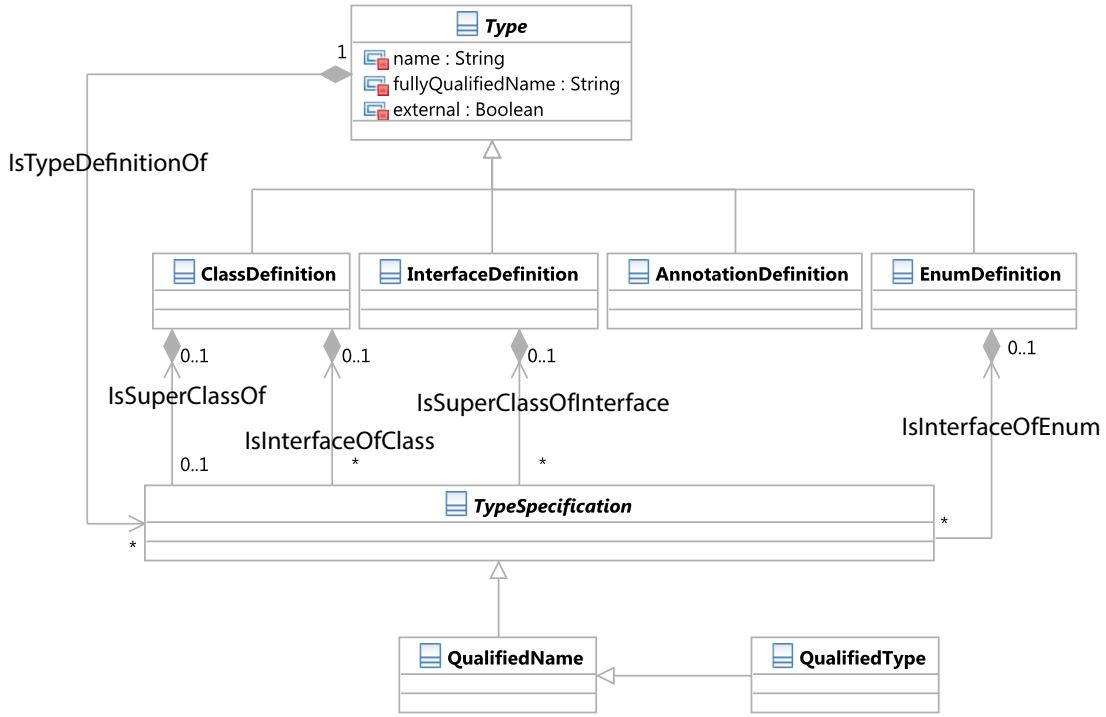

Figure 17: Extract of the TGraph schema to represent a Java software system [3]

An extract of the TGraph schema for the Java source code representation is given in figure 17. The complete Java TGraph schema contains around 90 nodes and 160 edges, covering the syntax of Java version 6 [3].
The Java to TGraph transformation functionality has been used to evaluate whether the implemented model difference calculation approach is capable of handling a completely different use case than difference calculation for a model versioning system. When sur-

veying existing model difference calculation tools and frameworks, UMLDiff [12] was investigated although it serves a very domain-specific purpose. UMLDiff is used to detect structural differences between versions of a software systems, represented by a domain-specific model type. The metamodel used by UMLDiff is very similar to the Java TGraph schema used by the Java fact extractor. As GDiff has been designed to handle models on TGraph level, it can now be tested whether the implemented approach is also capable of detecting structural differences of software system versions.
Therefore, GDiff has been tested on Java TGraphs that have been created using the webservice provided by the SOAMIG project. As the implementation of GDiff has been versioned in a Subversion [5] repository, two revisions of the implementation could be retrieved. These two version represent two major milestones in the development, the first being capable of detecting differences and generating a basic delta, the latter being capable of actually generating the correct delta in the required delta description language.
The source codes for both revisions as well as all dependencies to third-party libraries have been provided to the webservice, resulting in two Java TGraphs, each representing one of the revisions. The TGraphs are of considerable size, the first version containing 7122 vertices and 10870 edges, the latter containing 15238 vertices and 23786 edges. Vertex and edge count approximately have increased by factor 2. Due to the fact that the resulting models are given as TGraphs, they could directly be passed to GDiff to calculate the difference.
To achieve TGraphs that are actually processable by GDiff, their schemas have to contain certain elements required to generate the desired delta output conforming to the delta description language given in section 2.4. Specifically, the metainformation element containing the highest delta variable previously used must be contained in the model. Also, every model element must have an attribute containing a delta variable so that variables can be kept throughout deltas. As direct access to the schema of the Java TGraphs is not given and a modification of the schema and the TGraphs on the file as well as the JGraLab internal representation was not possible, GDiff had to be modified to be capable of handling Java TGraphs.
This resulted in a different development branch of GDiff, solely added to process these graphs. The requirement of metainformation vertices and attributes has been removed from the reduced implementation. As a consequence, the delta output has been altered because the desired delta description language could no longer be generated. Therefore, the generated delta output simply lists the amount of added, removed and matched elements to have an indicator on how well the difference detection performs. This reduced functionality means that all requirements regarding delta output, including delta minimalism, correctness and completeness, cannot be fulfilled.
Due to the fact that several restrictions have been added to the schema for TGraphs processable by GDiff, as given when introducing the schema for reduced UML Activity Diagrams in section 4.2, even the reduced implementation is incapable of processing the Java TGraphs.
First of all, a containment order is assumed to be represented by the schema so that element types can be processed in a top-down manner, starting with containers and

traversing down to contained elements. The Java schema however contains non-abstract elements, that generalise all other element types. Therefore, as soon as one of these non abstract types is processed, all elements from the host model are gathered for investigation. This leads to the following situation: If the latter version of the software system is set as host model, the first list of elements to be investigated contains all 15238 vertices of the host model. Hence, the goal to reduce element comparisons by assuming a hierarchical type order fails on the Java TGraph.
Furthermore, association elements, as for example Control Flows in the introduced subset of UML Activity Diagrams, are required to be defined as a vertex, connected to its source and target by exactly two edges. In the Java schema, such associations are directly represented by edges and therefore not processable by GDiff.
Add last, the structural similarity algorithm implemented in the assumption that associations between elements are represented like described above, results in way to high similarities whenever two elements are compared. The assumption results in the fact that a vertex can only be connected to other elements via three types of edges at maximum:

1. Connection to the target element or being the target element of an association
2. Connection to the source element or being the source element of an association
3. Aggregation connection to a container or a contained element

The similarity algorithm has been implemented, so that whenever a host element without any connections of one of these three types is compared to a candidate element that also is not connected to any element via that type returns a partial structural similarity of 1. This assumption works as expected if only these three types of connections are possible. Additionally it returns structural equality for elements that are not connected in any way.
The Java schema instead allows many types of edges between vertices. Whenever one element is not connected via most of the possible edge types, the structural similarity increases because the partial structural similarity for each unused edge type is 1. Hence, the resulting structural similarity is greater than 0.9 for most of the compared elements. Additionally, elements without a name element are only matched on the foundation of their structural similarity.
Thus, all elements from the target model are being matched to elements in the host model because in this specific test case the target model (7122 vertices) contains less vertices than the host model (15238 vertices).
As a result, GDiff is capable of processing the generated Java TGraphs but does not return the desired results. No conclusions can be made about structural changes between the two inspected versions which was the purpose of this investigation.
This would however be possible if the generated Java TGraphs would be transformed into TGraph with a schema that conforms to the schema restrictions made before. Resulting, for instance, in edge types of the Java TGraph converted to vertex types with edge connections to source and target.
The problem of type generalisation which results in all host model elements being returned as the first elements to be investigated, shows that the derivation of a hierarchical

type order cannot be assumed to work on any model type. The requirement (9) to outperform runtime complexity $O(n^2)$ is not satisfied for the given models but instead is exactly $O(n^2)$ and thereby still in the polynomial class which satisfies requirement (8). Although many of the requirements are not being satisfied in this specific test case, some of them could be satisfied. Necessary optimisations to achieve that will be described in the following section.

5.3. Required Implementation Improvements based on TGraph Evaluation

The evaluation of GDiff on Java TGraphs showed that some modifications to the implementation would improve its generality.
The underlying model representation, given by TGraphs, is being used to have a generic representation that can be used for any model type. As a matter of fact, the usage of TGraphs in the way it has been done in this work, builds a major restriction to processable models. Based on the evaluation of Java TGraphs, the metamodel of the processed TGraphs must conform to certain restrictions which where fulfilled for the subset of UML Activity Diagrams but not for Java TGraphs. For instance, association elements have to be represented by vertices, connected to their source and target by one edge each. These restrictions actually require a meta-metamodel for GDiff-processable TGraphs. Based on the assumption that every model could be transformed to conform to this meta-metamodel, the required generality of GDiff is still fulfilled.
Furthermore it appeared that a TGraph schema might have non abstract type generalisations. Although GDiff aims to only compare elements by type level to reduce the required amount of element comparisons, such generalisations make this reduction obsolete. This problem cannot be solved immediately and would require further investigation. Instead, it provides an enhancement to element comparison: Type similarity might also be considered for elements that share the same supertype.
Regarding similarity computation, name similarity alone, sequentially combined with structural similarity does not result in acceptable similarity values if trying to match elements. Because compared elements might be unnamed, name similarity cannot be used and therefore only structural similarity is considered. Whenever no elements have been matched before, structural similarity is however incapable of detecting new matches on its own depending of the used similarity threshold value.
Instead of only considering name attributes a generalisation of element comparison to attribute similarity might improve the matching mechanism. Such an algorithm would compare all attributes of the compared elements that occur in both of the elements. The similarity must however be specific to the type of the processed attribute, whether it is for example from the string or numerical domain.

6. Conclusion

This work provided a general solution to model difference calculation as intended. The embedding of such an algorithm as well as its requirements were introduced in section 2. A survey of existing algorithms, tools and frameworks in section 3 provided the general algorithmic solution which was described and implemented in section 4. Finally, the implemented prototype has been tested against UML Activity Diagrams and Java TGraphs in section 5.
Evaluation of the algorithm and its implementation turned out to be working as expected, fulfilling all the requirements that have been made.
Some optimisations, applicable to the conducted implementation have been given before, based on certain limitations resulting from the implementation. Some more general improvements, enhancing performance and generality of the approach still require further investigation in future works.
The general algorithmic solution to model difference calculation heavily relies on the chosen value of the similarity threshold. For GDiff, the value has been chosen on the foundation of certain aspects of the introduced metamodel for UML Activity Diagrams. The similarity threshold however actually needs further investigation to determine what an optimal value would be or whether it depends on the processed model. Such an investigation is out of scope for this work but it would certainly enhance the general process of model difference calculation.
While conducting the investigation of existing frameworks, SiDiff [2] appeared as most elaborated. Due to the fact that it has constantly been developed and enhanced since 2004, advanced methods for runtime optimisation have already been implemented. Similar enhancements could also be applied to the GDiff implementation to further improve its runtime complexity.
Finally, the investigation of delta generation in section 2.5 showed, that model difference calculation could be mapped to delta difference calculation instead. Hence, delta difference calculation tools could be used to identify differences between modelling deltas which would in turn represent the modelling difference. This approach would presumably be applicable to other delta domains as well and therefore is of great interest for further investigations.

A. Manual for GDiff

GDiff can either be run in single model mode, resulting in only a creation delta for the provided model, or full mode, resulting in a creation delta for the host model and a backward delta leading from host to target model. See the following manpage on how to set the startup parameters:

```
usage: java -jar gdiff.jar -n <filename> -c <filename>
            [-o <filename>] [-b <filename>]

-n <filename>   (Required) The tg file that contains the TGraph that
                represents the newer version of the compared model.

-c <filename>   (Required) Write the delta that if applied to the empty
                model results in the newer version of the model to
                <filename>. ("Creation Delta")

-o <filename>   (Optional) The tg file that contains the TGraph that
                represents the older version of the compared model.
                If this parameter is set, the parameter -b is required
                too.

-b <filename>   (Optional) Write the delta that if applied to the newer
                version (-n) of the model results in the older version
                (-o) of the model. If this parameter is set, the parameter
                -o is required too.
```

Listing 8: Manpage for GDiff

B. Manual for GUUID Applier

The utility GUUID Applier can be used to propagate delta variables from a creation delta to the corresponding TGraph. See the following manpage on how to set the startup parameters:

```
usage: java -jar gUUID-Applier.jar -t <filename>
            -d <filename> -o <filename>

-t <filename>   The .tg file that contains the TGraph which is
                to be enhanced with UUIDs.

-d <filename>   A delta file (.gd) containing a operation-based
                delta that generates the TGraph given as -t parameter
                if applied to the empty model. Delta is only allowed to
                contain add-operations.

-o <filename>   Write the resulting TGraph to the given file.
```

Listing 9: Manpage for GUUID Applier

References

[1] IBM. Rational Software Architect Website. Last Access: 20.11.2012. [Online]. Available: http://www-142.ibm.com/software/products/de/de/ratisoftarch

[2] P. Pietsch, "The SiDiff Framework Technical Report," 2009.

[3] T. Horn, A. Fuhr, and A. Winter, "Towards Applying Model-Transformations and -Queries for SOA-Migration," in *MSI 2009*, 2009.

[4] J. Vaspermann, *Essential CVS*. O'Reilly, 2003.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion. Next Generation Open Source Version Control*. O'Reilly Media, 2004.

[6] J. Loelinger, *Version Control with Git*. O'Reilly Media, 2009.

[7] J. Mukerji and J. Miller, "MDA Guide Version 1.0.1," 2003, Last Access: 18.10.2012. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[8] OMG, "XMI Mapping Specification Version 2.4.1," 2011, Last Access: 23.10.2012. [Online]. Available: http://www.omg.org/spec/XMI/

[9] Visual Paradigm. Visual Paradigm for UML 10 Website. Last Access: 20.11.2012. [Online]. Available: http://www.visual-paradigm.com/product/vpuml/

[10] Visual Paradigm. VP Teamwork Server 10 Website. Last Access: 20.11.2012. [Online]. Available: http://www.visual-paradigm.com/product/vpts/

[11] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference Computation of Large Models." ACM Press, 2007, pp. 295–304. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1287624.1287665

[12] Z. Xing and E. Stroulia, *UMLDiff: An Algorithm for Object-Oriented Design Differencing*. ACM, 2005, no. 6, pp. 54–65. [Online]. Available: http://portal.acm.org/citation.cfm?id=1101919

[13] Y. Lin, J. Gray, and F. Jouault, "DSMDiff: A Differentiation Tool for Domain-Specific Models," *European Journal of Information Systems*, vol. 16, pp. 349–361, 2007.

[14] Object Management Group. (2011) OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1. Last Access: 11.11.2012. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[15] A. Winter and D. Kuryazov. (2013) GMoVerS Project. Software Engineering Group, Carl-von-Ossietzky-Univerity Oldenburg, Germany. Last Access: 08.06.2013. [Online]. Available: http://www.se.uni-oldenburg.de/preprocess.php?seite=45623.html&include0=generated-content/ModelingDeltas.html

[16] D. Kuryazov, J. Jelschen, and A. Winter, "Describing Modeling Deltas By Model Transformation," in *Softwaretechnik Trends (Issue on International Workshop on Comparison and Versioning of Software Models (CVSM 2012))*. Gesellschaft für Informatik, to appear 2012.

[17] D. Kuryazov, A. Solsbach, and A. Winter, "Towards Versioning Sustainability Reports," in *5. BUIS-Tage: IT-gestütztes Ressourcen- und Energiemanagement*, no. to appear. Springer Verlag, 2013.

[18] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "A Metamodel Independent Approach to Difference Representation," *Journal of Object Technology*, vol. 6, no. 9, pp. 165–185, October 2007. [Online]. Available: http://www.jot.fm/issues/issue_2007_10/paper9/

[19] M. R. Garey and D. S. Johnson, *A Guide to the Theory of NP-Completeness*, ser. Computers and Intractability. W.H. Freeman and Company, 1979.

[20] M. Alanen and I. Porres, "Difference and Union of Models," in *Proc 6th Int. Conf. on the UML*, P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863. Springer, 2003, pp. 2–17.

[21] J. W. Hunt and M. D. Mcilroy, "An Algorithm for Differential File Comparison," *Computing Science Technical Report*, no. 41, pp. 1–9, 1976. [Online]. Available: http://www1.cs.dartmouth.edu/~doug/diff.ps

[22] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic Differencing for Activity Diagrams," 2011.

[23] K. McMillan, "The SMV System," Tech. Rep., 2000, Last Access: 13.12.2012. [Online]. Available: http://www.cs.cmu.edu/~modelcheck/smv.html

[24] S. Maoz, J. Ringert, and B. Rumpe, "CDDiff: Semantic Differencing for Class Diagrams," in *Proc. 25th European Conference on Object Oriented Programming (ECOOP'11)*, 2011.

[25] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[26] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models," in *Software Engineering*, 2005, pp. 105–116.

[27] J. Ebert and A. Franzke, "A Declarative Approach to Graph Based Modeling," in *Graphtheoretic Concepts in Computer Science, number 903 in LNCS*. Springer, 1995, pp. 38–50.

[28] S. Kahle, "JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen," Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[30] Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR). (2013) SOAMIG - Migration von Legacy-Softwaresystemen in serviceorientierte Architekturen. Last Access: 05.06.2013. [Online]. Available: http://www.soamig.de/

[31] A. Fuhr, V. Riediger, and T. Horn, "An Integrated Tool Suite for Model-Driven Software Migration towards Service-Oriented Architectures," *Softwaretechnik-Trends*, vol. 31, no. 2, 2011.

[32] A. Baldauf and N. Vika, "Java-Faktenextraktor für Gupro," 2008.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.