automated sequential composition of deltas and related optimization operations

An additional research to metamodel independent difference representation
# Table of Contents

Table of Contents .......................................................................................................................2

1 Introduction ..........................................................................................................................3
1.1 Outline of the Thesis ..................................................................................................3

2 Basic Concepts and Tools .....................................................................................................4
2.1 Model-Driven Engineering (MDE) ..............................................................................4
2.1.1 Domain Specific Modeling (DSM) ........................................................................4
2.1.2 Model-Driven Architecture (MDA) .......................................................................6
2.1.3 Meta-Object Facility (MOF) ..................................................................................6
2.1.4 Kernel MetaMetaModel (KM3) ..............................................................................7
2.2 Transformations .........................................................................................................8
2.2.1 ATLAS Transformation Language (ATL) ............................................................9
2.2.2 Higher Order Transformation (HOT) ..................................................................11
2.3 Model Weaving ........................................................................................................11
2.3.1 ATLAS Model Weaver (AMW) .........................................................................12
2.4 Software Evolution ...................................................................................................13

3 Differences .........................................................................................................................15
3.1 Difference Representation .......................................................................................16
3.2 Meta-Model Independent Approach ........................................................................17
3.2.1 Difference Metametamodel (MMD) ....................................................................17
3.2.2 Constructing Constraints/Rules ...........................................................................18
3.2.3 Example ................................................................................................................19
3.3 Difference Application ..............................................................................................20
3.4 Operators .................................................................................................................21
3.4.1 Dual Calculation ....................................................................................................22
3.4.2 Sequential Calculation ..........................................................................................22
3.4.3 Parallel Calculation ...............................................................................................23

4 Sequential Calculation ........................................................................................................24
4.1 Concept ....................................................................................................................24
4.1.1 Constraints ...........................................................................................................25
4.2 Cases .........................................................................................................................26
4.2.1 Existing ..................................................................................................................26
4.2.2 Non Existing .........................................................................................................27
4.2.3 Examples ................................................................................................................27
4.3 Implementation ........................................................................................................29
4.3.1 First Problem: adding or deleting of contained elements ...................................29
4.3.2 Second Problem: changing an contained element ..............................................31
4.3.3 Testing ..................................................................................................................31

5 Personal Conclusion ...........................................................................................................33
5.1 Changing the Problem ..............................................................................................33
5.2 Future Work ...............................................................................................................35

Acronyms ..................................................................................................................................36

Reference .................................................................................................................................37
1 Introduction

Since programmable computers became common, developers tried to raise the level of abstraction, making it easier to develop software on the one hand and avoiding errors in the developing phase on the other hand. This evolution of software engineering underwent several main steps, like the change from machine code to Assembler (2nd level programming language) to high level programming languages like FORTRAN or COBOL and later to PASCAL, C and Java. Model-Driven Engineering (MDE) was the next step in raising the abstraction. Combined with the concept of domain-specific modeling (DSM) it can increase productivity by 500-1000%, which happened last when there was a change from Assembler to high level languages. Today in use by companies like Nokia, it is still a rich area of research.

To gain the most benefit of using models for software developing there’s a need for standardization. For software evolution purposes (and more) it is necessary, to not only handle models, but also the differences between them. In 2007 Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio presented a metamodel independent approach to difference representation. In the year 2008 Antonio Cicchetti presented his doctor thesis, another paper with rich research on this topic. A sub theme of this paper is the sequential composition of those differences. This thesis presents an approach to sequential calculation, hence it can be considered as an enhancement/enlargement to Cicchetti’s work. According to this, these two papers are normally not mentioned as references in the text. References to other papers are done either by square brackets (e.g. [1]) or by superscription (e.g. 1). All other numerical references refer to chapters in this text.

1.1 Outline of the Thesis

The thesis starts with a broad overview of basic concepts, like Model-Driven Engineering (MDE), Transformations or Software Evolution. One focus is on ATLAS Transformation Language (ATL), the language later used to implement this approach. The second chapter of the thesis is about Differences with Cicchetti’s Meta-Model Independent Approach as the main focus. Also Operators are introduced, including Sequential Calculation, which is the next chapter. Once again, the Concept is explained and some Constraints are described, which need to be considered. Afterwards, the possible sequential calculations are divided into Cases and an Implementation approach is presented. Finally the Personal Conclusion of the author is presented, including an approach to solve several problems which occurred through the implementation research and a brief foresight on this topic.
2 Basic Concepts and Tools

In this chapter some of the basic concepts used by this thesis are outlined to inform readers who have no or only little contact to Model-Driven Engineering. In section 2.1 the whole concept of software modeling (MDE) and the techniques used (MOF, KM3) are described. Section 2.2 introduces the concept of model transforming and ATL, the transformation language which is used in this thesis. Section 2.3 is about model weaving. Finally in section 2.4 the concept of software evolution is introduced.

2.1 Model-Driven Engineering (MDE)

In traditional code-driven software engineering, models where often not anything more then personal drawings, desultory done for their own purpose by some developers, confined to a simple documentation role instead of being actively integrated into the engineering process. Since models were always a vehicle for communication, they became rapidly an effective way to exchange all kind of information (design, interfaces, requirements etc) by all kind of developers (business process engineers, control engineers, computer scientists etc). MDE raises models into chief position of the designing phase in software engineering and so raises the abstraction grade of software designing.

In a first step models of the system and its components are designed to ease the communication, but also to provide an additional basis for pre-implementation, validation of requirements and quality properties as well as for automatic generation of source code. In a second step the code is written manually, based on the constructed models (maybe supported by some automatisms).

The idea behind MDE is to find an easy, clean and productive solution for the designing phase of software engineering which is abstracted from the complexity of the implementation. It also provides some techniques for formal design checks (e.g. HLPNs) and eases the use of patterns since they are often model-based.

Bézivin declares in his paper “everything is a model”. Compared to the old approach: “everything is an object” this means that all structures can be represent as models leveraging models into first class position in all kinds of development. Some advantages of this idea are highlighted in this paper.

2.1.1 Domain Specific Modeling (DSM)

After several years of model driven engineering the lack of a universal approach became obvious: General Purpose Languages were too broad. Automatic implementation based on the designed models was not possible (although there are some approaches based on UML profiles, called Excutable UML or xUML) and so the coding was done mostly manually. In the implementation phase modifications of the system only appeared in the code, because of high maintenance costs. Corresponding models were left unchanged, leading to not updated, invalid - and hence useless - models. The obvious benefit of this technique is very low compared to the costs.

In DSM a solution is created for a narrow field of application called a domain. According to Kelly and Tolvanen a domain can be defined as an area of interest to a particular development effort. Domains can be a horizontal, technical domain, such as persistency, user interface, communication, or transactions, or a vertical, functional, business domain, such as telecommunication, banking, robot control, insurance, or retail. DSM raises the level of
abstraction beyond current programming. The narrower a domain is, the more possibilities for automation we get, hence domains often consist of smaller sub domains.

A DSM consists of three parts:

- A domain-specific modeling language (DSL)
- A domain-specific code generator
- A domain framework

The DSL is a special language, designed or adapted to characterize a domain. A DSL is a set of coordinated models. Each of those models specifies one of the following aspects:

- Domain definition metamodel (DDMM): introduces the basic elements of a model (such as places, transitions and arcs in a PetriNet) and their relations.
- Concrete syntaxes: transformations, which map the DDMM onto a display surface metamodel (such as circles for places in the PetriNet example)
- Execution semantics: may also map the DDMM to another DSL with an execution semantic or even to a GPL (for example, mapping the firing rules of the PetriNet to Java code)

The domain-specific generator generates – based on one or more models constructed in the correlating DSL – code which runs on the correlated domain framework. Since the coding is done by the code generator, it’s sufficient, if this part of the DSM is created by high skilled programming experts. Later on, the models, done in the DSL, can be produced by nearly anybody, which makes it easier for domain experts to develop the software. In DSM the main aspect of the engineering is located on the modeling and – in contrast to the former – stays on the model. Obvious a lot of advantages arise, for example:

- Lower costs – since we need only few high paid experts to construct the framework and the code generator
- Better Quality – since the models can be checked for errors automatically (for example based on OCL) and the coding is done automatically through code generators which are optimized by high skilled engineers.
- Higher Range of developers – DSM distinguishes two roles: those who create the DSM and those who use it. Mostly DSL are easier to handle, hence even developers with no knowledge in coding but maybe more experience in the domain can produce the software.
- Higher Productivity – in the long term the designing productivity increases but the largest benefit takes place in maintaining the software.

Since most companies are acting in small business segments, DSM is ideal for them. Of course, DSM is not the perfect solution for all companies and cases.
2.1.2 Model-Driven Architecture (MDA)

In 1996 the OMG started to expand their scope to modeling. Belonging development processes based on abstractions have been standardized and proposed by OMG under the name of MDA. Its main target is to separate the specification functionality from the specification of the implementation to solve the three primary goals which are portability, interoperability and reusability through architectural separation of concerns. This is done in three steps. First of all, a Computation Independent Model (CIM) is constructed. This is later extended to a Platform Independent Model (PIM) which already uses a general-purpose, platform independent modeling language such as UML. Finally this model is transferred into a Platform Specific Model (PSM), which focuses on the implementation details of a certain platform.

MDA promises many benefits namely reduced costs throughout the application life-cycle; reduced development time for new applications; increased return on technology investments (ROI) and rapid inclusion of emerging technology benefits into existing systems.

2.1.3 Meta-Object Facility (MOF)

A model represents a system. To define a model it is necessary to define a metamodel, which describes all contained model elements, and the way they are arranged, related, and constrained. As models consist of elements, metamodels do too. The model elements are related to their conforming metamodel elements by meta-relations. So the element is typed by its metatype. A model conforms to a metamodel if and only if each model element has its metatype defined within the metamodel. Obviously a lot of different metamodels can exist (UML, Petri Nets etc). This growing number makes it necessary to define the way, in which they are represented. One solution is a metametamodel, dedicated to the definition of metamodels. As models are based on metamodels, metamodels are based on a metametamodel. Since a metametamodel consists of elements too, a solution to represent metametamodels is needed again. To avoid a vicious circle, the OMG did introduce with MOF the four-level architecture. The MOF metametamodel is defined recursive, so it only exists of elements, which are typed by the MOF metametamodel itself. A typical solution based on the MOF four-level architecture exists of three layers of models:

- M3: the metametamodel
- M2: the metamodels which conforms to the metametamodel
- M1: the models which each conform each to a metamodel
- An additional layer M0 represents the existing instances of the models
2.1.4 Kernel MetaMetaModel (KM3)

KM3 (Kernel MetaMetaModel) is another meta-modeling language, founded by INRIA. It's based on the same concepts as MOF, for example the self-related property. KM3 focuses on meta-modeling concepts and lacks some features of other meta-modeling languages, for example it doesn't support Java code generation like the Ecore model. The use of KM3 is mainly justified by its simplicity and flexibility to write meta-models and to produce Domain-Specific Languages (DSLs). A lot of KM3 based metamodels are already preproduced and collected in the Atlantian Zoo.

KM3 is a DSL to define metamodels. The domain definition metamodel is the KM3 model. The concrete syntax is a textual syntax defined through INRIA. The semantic of KM3 is also defined and enables the specification of metamodels and models. KM3-models can easily be mapped to MOF and Ecore which makes KM3 usable with programs like Eclipse.

MOF is often called a meta-modeling language. Those languages mostly are only distinct by small differences since they are generally based on the object oriented idea. (Meta)classes can contain attributes and are connected through references, which are also used for inheritance and typing. These meta-modeling languages generally only represent the semantic part. There are other languages to describe how to visualize the models and metamodels.
2.2 Transformations

A lot of different levels of abstraction and different kinds of models were mentioned in the last chapters. But there is something needed to glue all this together. Transformations make it possible to transform one or more models, the source, into other one or more other models, the target. The metamodel of source and target can be the same; in this case the transformation is called exogenous. If they differ the transformation is called endogenous. A transformation contains all necessary information to map the source to the target in transformation-rules, which are based on a MTL (for example MOF/QVT or ATL). The MTL itself is based on a metametamodel and correlates with the metamodels of the source and the target which are based on the same metametamodel. A transformation engine later on uses the rules defined through the MTL to create the target model(s) out of the source model(s). A very important distinction can be made between model-to-model transformations and model-to-code transformation. Though it seems like a subset (because code can also be represented as a model) there is a momentous difference: while the former is a transformation on the same level of abstraction, the later changes this level. A good overview of transformations and some examples can be found in [17].

Transformations play a significant role in development автоматisms. Especially as there are a lot of different metamodels used, MDE would be more inconvenient without it (I need a KM3-based model, you give me an Ecore-based model? No Problem, it is transferred with one click).
2.2.1 ATLAS Transformation Language (ATL)

Conforming to INRIA ATL is a transformation-based model management framework, with metadata management and data mappings as the main applications. It is general on a high grade of abstraction, making it possible to map to many different languages. It is consistent with many standards, especially the MDA MOF/QVT. ATL was released in 2004 as a part of AMMA (further information see [11]) under the Eclipse Public License which makes it Open Source. ATL comes with some Tools called ADT (similar to JDT), for example an Eclipse IDE, including and editor with syntax highlighting, an outline and simple wizards, and a debugger. Further ADT includes the ATL Engine. A compiler produces byte code (.asm) out of an ATL-Project which can be executed on a virtual machine. The virtual machine can run on top of different platforms. Since ATL is used in this paper, it is focused on this transformation language.

An ATL Transformation is based on one or more ATL files (.atl):
- Module (the standard type, a normal module-to-module transformation)
- Query (as the name implies a module-to-primitiveDataType transformation)
- Library (can be imported by any type of ATL File including libraries itself)

An ATL Module consists of four parts: the header, the import, helpers and rules. They are described in the following paragraphs.

```java
module module_name;
create output_models [from|refines] input_models;
```

The **header** defines the name of the transformation and the input as well as the output models (also known as source and target models). The syntax for the input and output models is `[variable name] : [metamodel name]`. It is only possible to navigate through source models (which are consistently read only while target models are write-only).

The refine mode, enabled by the word “refines” (in contrast to “from”), allows the developer to save code, since unchanged elements are automatically copied from the source to the target. Corresponding to [22] section 3.1.2.2, the refine mode can only be used with a single source and a single target model which need to conform to the same metamodel.

```java
uses extensionless_library_file_name;
```

The **import section** makes it possible to import various libraries. Later on, the imported helpers can be used in the module.

```java
helper [context context_type]? def : helper_name(parameters) :
return_type = expression;
```

**Helpers** can be conceived as the equivalent to Java methods. Their name can be compared to the name of the Java method, the return type defines the type of the return value (note: there is nothing like “void”). An ATL expression defines the computing part of the helper. The helper can also contain parameters, which are coded “[parameter name] : [parameter type]”. Since helpers can be invoked by other elements, the context describes the type of element, a helper can be called by, later accessible as “self”. If there is no context, the helper is invoked in context of the module itself. Helpers are deterministic and can be recursive.
The ATL expression syntax is very close to the OCL syntax. Further information can be found in [22] section 4.

If the parameter section is omitted, the helper is called “attribute”, something similar to a context specific constant. The main difference between attributes and helpers is that helpers are executed every time they are invoked. Attributes are only executed the first time they are invoked and later on only return the first calculated value.

**Rules** are the heart of a transformation. The typical rule is called “matched rule”. Such a rule primarily matches a source pattern over source models and a target pattern that creates new elements in target models for every match. They comprise of four sections and although the “from” (source) and the “to” (target) sections are necessary the “using” and “do” sections are optional. The “from” section specifies the source element. The optional condition part makes it possible to distinguish between elements of the same type. Completing the manual, it is possible, to use elements of two different source-models in the “from” section.

```
module module_name;
create OUT : MM from IN1 : MM, IN2 : MM;

rule sample_two_input {
    from
        a : MM!Element in IN1,
        b : MM!Element in IN2 (  
            compare condition e.g.: a.name = b.name  
        )  
    ...
}
```

The “using” section provides us with the opportunity to declare variables to use them later on in the rule.

The “to” section finally makes it possible, to create elements in the target models. It defines an arbitrary amount of target-elements and their bindings. The bindings can refer to the element(s) introduced in the “from” section. It is also possible to define a set of target model elements by using the word “distinct” and the corresponding syntax.

The “do” section makes it possible to define some additional imperative statements, which are run after the target model initialization has been finished. I don’t give their syntax in this paper.
If a matched rule is introduced by the word “lazy”, it has to be invoked by another rule. If it is introduced by the words “unique lazy” it can only be invoked once.\textsuperscript{21}

In contrast to matched rules, “called rules” are only invoked by do-statements. There are two main syntax differences to normal rules: they have parameters and they don’t have a from-part. If called rules start with the word “entrypoint”, they are invoked at the beginning of the transformation.\textsuperscript{22} Navigation is performed using OCL expressions.

As you can see, an ATL-transformation consists of declarative and imperative elements (do-statements and called rules) making it a hybrid technology, although it is highly recommended to use the declarative techniques whenever it is possible since they enhance the readability and matches better modeling.\textsuperscript{23}

Libraries contain several helpers and can be reused in modules, queries and even other libraries. They start with the expression \texttt{library [library name]} followed by an arbitrary amount of helpers. Queries are an easy way to extract primitive data from models, for example used to export strings. They are declared in the following way:

\begin{verbatim}
query [query_name] = [expression];
\end{verbatim}

\subsection*{2.2.2 Higher Order Transformation (HOT)}

Let’s reach another stage of abstraction: transformations are said to work on models. But we can think of transformations themselves as models, holding on the modeling paradigm “everything is a model”.\textsuperscript{24} Transformations which have transformations as input and/or output are called HOT. They offer us a complete new prospect. For example it is possible to create metamodel independent transformations, which create a metamodel dependent transformation through a HOT based on a special metamodel as input. This technique is used in 4.3.

\section*{2.3 Model Weaving}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{weaving_model.png}
\caption{a weaving model is used to connect two models\textsuperscript{5}}
\end{figure}

Model weaving is on another level of abstraction to transformations. The function of weaving models is to set precise correspondences between models or metamodels. A weaving model consists of typed links between model elements (it is possible to weave more
than two models or metamodels, but without loss of generality this paper relates to two metamodels. Differing from transformation, weaving models don’t imply a change from one model to another model creating a new one, but rather indicate a semantic relation between the corresponding models. In consequence, mapping information can be stored relatively simply, compared to a transformation language.

A typical usage of weaving models is shown in Figure 2.3. A weaving model maps two metamodels LeftMM and RightMM to each other, implying a fine grained relationship between those two metamodels. The weaving model conforms to a weaving metamodel which itself conforms to the same metametamodel as the connected metamodels. Since the weaving model is based on a metamodel, it is possible to use it as input for a transformation. So the weaving model is not much more then a set of links between the two metamodels. In general, those links can’t be generated automatically because they are often based on human decisions, but in the most cases there are heuristics to approximate the result. Those heuristics check the signatures of elements, including name, kind and type. If there are several matches, structural similarity is used to construct the weaving model. Later, those automatically generated models can be fine-tuned by manual optimization. It’s even possible to match elements, which are normally not related, giving the developer extensive possibilities, for example pattern-driven evolution.

2.3.1 ATLAS Model Weaver (AMW)

AMW is a component-based platform for model weaving, i.e. establishing and managing correspondences between models. Simply said, it’s an IDE for Eclipse. After uploading the weaving metamodel, the GUI is established. Then the LeftMM and the RightMM can be chosen and the links can be established.

Figure 2.4: Screenshot of the AMW
2.4 Software Evolution

Everybody who ever developed software knows that it is impossible to write correct and optimal software. Furthermore software which is correct in the eyes of the developer can fail in the eyes of the user. Today, most of the software is passed through many levels of development, not only the different phases of software engineering but also many different version identified by more or less small changes.

This software evolution induces high costs, making it worthwhile to have an intensive and accurate design phase before implementing the software. As mentioned in 2.1, this approach was often rejected because of short-termed higher costs in the designing phase, leading to an error driven development in the 90's. Nevertheless, even with an extensive design phase, it is not possible to create perfect software without implementing prototypes and even final versions, always including failures and a need to be maintained leading to a lot of different versions of a system. This incremental software evolution is called sequential evolution. And can be thought of as the vertical dimension in a versioning tree. An example of this kind of versioning in everyday live is Wikipedia. If an article is updated by a user, the new version is saved, whilst the old version is kept in the system, making it possible to refer to a certain version of an article.

“Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software. Within a given version number category (major, minor), these numbers are generally assigned in increasing order and correspond to new developments in the software. At a fine-grained level, revision control is often used for keeping track of incrementally different versions of electronic information, whether or not this information is actually computer software.”

Today a lot of different developers are working on the same projects, bringing us to the next problem: the support for collaborative work or multi user access, resulting in a lot of different versions on the same vertical level, called parallel evolution. (the horizontal dimension in a versioning tree). Necessarily special developing tools are needed to keep on track with the different versions, making it possible for several developers to work on the same parts of a project. Systems which solve this requirement are called Version Control System (VCS) or Source Code/Control Managementsystem (SCM). Those systems store the different versions on a centralized server, called the repository, making it possible to step back to any earlier state. Developers check out the actual version as their working copy. The local version can be held current by incremental updates from the repository. If the local working copy has reached a new level, the developer can commit his version to the repository (VCS do offer many further features, which are not mentioned in this paper).
Some problems do appear, for example there is a need to handle the fact that several developers may want to work on the same data or files on the same time. One approach is the pessimistic revision control, meaning the file is locked while somebody is working on it, making it impossible to have two or more persons working on it at the same time. It is used by versioning software like RVS or VSS. Optimistic revision control (e.g. used by CVS or SVN) uses the fact that two or more changes in a file might not influence each other. If a user tries to commit a file which was changed meanwhile, the system detects a version mismatch. In this case, there are three ways to handle the situation: either to throw away one of the two versions or try to merge them manually (supported by some tools) or automatically by heuristics (sometimes it is not possible to match them and a conflict arises). The third possibility is to start a new branch, meaning a new parallel version-tree, different to the so called trunk, which is the main developmental sector. For the sake of completeness it has to be mentioned that versioning and multi user access can be found in all kind of software and media (for example hypermedia documents).
3 Differences

As explained before, Software can exist at different versions, which are stored in a VCS. As the word “different” implies, these versions are only separated by small differences. Differences are the way in which two or more things which you are comparing are not the same. Since it is contrary to equality, a change between two versions is needed. Storing the complete information to every version would be redundant; hence only one complete version and the differences between this and the other versions are stored. It’s one constraint of differences, that you can use the stored version V1 and the difference to another version V2 to calculate the complete version V2. And while this approach is transitive, it is possible to just store the difference to any computable version.

Differences can appear in varying fields. For example, most backup systems only store the information and data which changed. Most of today’s software does deliver their updates as so-called patches, nothing more then the differences to the installed version. Another example is the partial refresh of the GUI of most software systems. The use of differences has several advantages. First of all the space needed to store differences, is quite obvious smaller than the space needed to store the complete version. Secondly, it eases the testing and selection of software. The developer can easily select modifications to detect which ones corrupt the system. Finally, it helps to better understand the evolution the software undergoes, since the changes between two versions become clear, while the storage of two different versions must be compared in their entirety.

Sticking to software evolution, there are two kinds of approaches to using differences. Forward differencing means, an initial version of the software is stored; later versions can be computed by the appliance of differences. Since this approach seems rational, there’s some benefit to the second approach of backward differencing. Contrary to the former, this approach uses a complete final version and differences to calculate the previous versions. The mentioned benefit lies in the fact, that the newest – therefore most accessed – version is always available. Furthermore, differences can be intensional, which means, they are based on a precise initial version, or extensional, which means, they can be applied to several revisions.

There are overall three ways of calculating differences:

- Textual: the two versions are treated like text using tools like diff or vdelta to calculate the longest common substrings and calculating differences based on inserts and deletes of strings. This approach does not support moving or changing operations.
- Syntactic: the two versions are compared in regard to their syntax, structure and their parse tree.
- Semantic: the two versions are compared on their semantic level, for example with Semantic Diff.

This thesis focuses on model driven engineering; hence it only surveys differences in models. Those models are mainly stored in XML (XMI) files. A structure-related calculation approach is needed. Also the problem of identifying objects has to be solved. In a two step approach (including the detection of element mappings in step one) is illustrated. A formal definition of model differenced versioning would be: \( M_1 \) and \( M_2 \) are models based on the same metamodel MM. The difference \( \Delta \) from \( M_1 \) to \( M_2 \) can be calculated, formalized as \( \Delta = (M_2 - M_1) \).
Some information about traditional techniques and constraints in difference representation is presented in section 3.1, leading right into the model-independent approach this thesis is based on in chapter 3.2. In section 3.3 some basic application ideas are presented and in section 3.4 further operators to manage and work with differences are described, including Sequential Calculation, which is discussed precisely within this thesis.

### 3.1 Difference Representation

Once, the differences are calculated, it is necessary to store them. To gain the largest benefit, the used method should be independent of the calculation method on the one hand and conform to some requirements on the other hand. Referring to [3], those requirements are:

- **model-based**, the outcome of a difference calculation must be represented as a model to conform to the spirit of “everything is a model” principle [12] and to enable a wide range of possibilities, such as subsequent analysis, conflict detection or manipulations;
- **minimalistic**, the difference model must contain only the necessary information to represent the modifications, without duplicating parts such as those model elements which are not involved in the change;
- **self-contained**, as a complement of the previous property, the difference must contain all the required information to autonomously represent the manipulations, without relying on portions of data contained in the compared models;
- **transformative**, each difference model must induce a transformation, such that when applied to the initial model it yields the final one. Moreover, the transformation must be applicable also to any other model which is possibly left unchanged in case the elements specified in the difference model are not contained in it. In other words, the transformation should give place to patch-like updates;
- **invertible**, each difference model must be invertible, such that whenever applied to the final model nullifies changes thus getting the initial model. Furthermore, all the information needed to obtain the inverse must be contained in the delta itself;
- **compositional**, the result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations;
- **meta-model independent**, the representation techniques must be agnostic of the base meta-model, i.e. the meta-model the base models conform to. In other words, it must not be limited to specific meta-models, as for instance happens for certain calculation methods which are given for the UML meta-model;
- **layout independent**, the proposed mechanisms must be agnostic of the presentation issues, i.e. the concrete syntax defined for the difference meta-model. In other words, the solution must be not limited to specific visualization approaches, like edit scripts, coloured diagrams or delta trees.

The mentioned requirements imply some qualities, for example do minimalistic, self-contained and invertible imply the Dual Calculation and model-based and transformative properties satisfy the “everything is a model” paradigm. Meta-model and layout independent leads the way to use the method in DSM. The transformative property implies
an automatic transformation $T_\Delta$ induced by $\Delta$ so that $M_2 = T_\Delta(M_1)$ is imperative (with $M_1$ and $M_2$ are models conforming to the same metamodel).

<table>
<thead>
<tr>
<th>Property</th>
<th>edit scripts</th>
<th>colouring</th>
</tr>
</thead>
<tbody>
<tr>
<td>model-based</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>minimalistic</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>self-contained</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>transformative</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>invertible</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>compositional</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>meta-model independent</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>layout independent</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 3.1

There are two major techniques to visualize differences, directed and symmetric deltas.28

Directed deltas can be compared to a step-by-step recipe to prepare the new model. Normally it’s a list of add, delete and edit commandos, which transform a given model into a new one. Since it’s linear and text-based, it is quite easy to optimize it or to compose it. On the other hand, the directed approach lacks in clarity and is pretty hard to read for a developer (which lowers the benefits of using differences). There are also some further disadvantages as use of direct identification, which makes this approach impractical for interoperable differences.

In contrast, symmetric deltas use a highly visual technique, representing the initial and the final model while coloring the changes. For example, all elements, which appear in the initial model but not in the final model, are colored red, showing that they are deleted. All elements, which did not exist in the initial model, but appear in the final model, are colored green, meaning they are added etc. Although this approach is highly readable, it is in no way minimalistic or easy to compose. So there’s a need to develop a new approach, which fulfills all the announced properties.

### 3.2 Meta-Model Independent Approach

It is necessary to find a general approach to model differences, which fulfill all the mentioned properties. Talking about “modeling” differences is possible, since we are only working with models, according to the MDE paradigm “everything is a model”. Staying on this, we can consider the differences have to be represented as a model, too. Models always conform to a metamodel, and so too must the differences.

A unified method to represent metamodels was already announced in 2.1.3, hence to have a unified and metamodel-independent approach to represent and store difference models, we need to have a method to construct difference metamodels out of arbitrary metamodels. This method is presented in this section.

#### 3.2.1 Difference Metametamodel (MMD)

Difference models are positioned on the level M1 of the four-level architecture. So, the difference metamodel has to be positioned on the level M2 (see Figure 3.1). To represent differences, every modeling element (conforms to the “class-type” of the KM3 model) of the original metamodel MM needs options to add or delete it. Furthermore, to not always delete and re-add an element, if one or more characteristics did change, a change-option is
needed as well. The easiest way to fulfill these needs is to construct a difference metamodel which is the original metamodel with every contained element $MC$ complemented with the following subclasses

- **AddedMC**, if a new element is added in the final model, which is not present in the initial model, it is done by using the corresponding AddedMC class in the difference model.

- **DeletedMC**, if an element is deleted, meaning it existed in the initial model but disappears in the final model, it is done by using the corresponding DeletedMC class in the difference model.

- **ChangedMC**, an element in the final model can differ from the corresponding element in the initial model. In this case, the old element is represented in the difference model in form of a ChangedMC element, which contains all attributes and references, which shall be deleted, and which relates via the reference $updateElement$ to an element of the MC class, which contains the new added attributes and references.

![Figure 3.1: Difference metamodel generation](image)

An ATL transformation to construct a difference metamodel out of any metamodel based on KM3 is given in [36]. The ATL transformation MM2MMD (Metamodel2MetamodelDiff.atl) builds a difference metamodel MMD by copying all elements of the original metamodel MM. Furthermore, three classes $AddedMC$, $DeletedMC$ and $ChangedMC$ are added for every non abstract class in MM. These new classes are subclasses of the original class MC on the one hand and of their respective abstract class $Added$, $Deleted$ or $Changed$, which are also added to MMD on the other hand. To handle ordered references, it is necessary to add the position index to the ordered classes (compare figure 4.9 in [3]).

### 3.2.2 Constructing Constraints/Rules

- **model-based, meta-model independent and layout independent**, easy to see, that the difference model is a model itself and that this approach is metamodel independent since it can be applied to all metamodels. Layout independence is given through the
fact that difference models can be easily calculated to any arbitrary visualisation like colouring or editing scripts.

- **minimalistic, self-contained, invertible**, difference models are by definition minimalistic and self-contained. For example, if a container is deleted, all contained elements need to be deleted, too, in the delta, since otherwise the self-contained and so the invertible properties would not be fulfilled.

- **transformative**, for every difference meta-model, a transformation can be automatically derived, so that the final model can be calculated by the use of the initial model and the difference model as input.

- **compositional**, is given, too. For example the Sequential Calculation is discussed extensively in this document.

### 3.2.3 Example

This is a simple example based on the simpleUML example, which can be found in [3] in chapter 4, Figure 4.6 and 4.8 (including larger examples).

#### Figure 3.2: Initial model $m_1$ of a person with a address

In Figure 3.2 an initial model $m_1$, based on the simpleUML metamodel is presented. It consists of a Class person, which has forename and lastName Attributes. It’s associated with a Class address, which consists of the Attributes street and city. This initial model shall be changed to a new model $m_2$. For some reason lastName is changed to surname and we don’t want to store the address anymore. Instead, we want to store contact information, for example the phone number. This model can be seen in Figure 3.3

#### Figure 3.3: final model $m_2$ of a person with contact information

Obviously, there are some changes, which need to be stored in a difference model. First of all, the Class c2 and the belonging Association ass are deleted. A new Class c2 and its belonging Association ass2 and its Attribute a5 are added. Finally, the Attribute a2 is changed. So let’s have a look at the difference model $Δ$ in Figure 3.4.
3.3 Difference Application

- Not only the Class `c2` is deleted but also its contained Attributes `a3` and `a4` need to be deleted, due to the self-contained property described in 3.1. This is necessary to enable the Dual Calculation described in 3.4.1.
- The class `c1` is also changed, although none of its properties are changed. This is necessary because one of its contained elements is changed and because a referenced Association is changed.

![Figure 3.4: difference model Δ](image)

### 3.3 Difference Application

After discussing the concept and the construction of differences, it’s necessary to point out the benefit of them: how to use them, to transform one model into another one. Looking at Figure 3.5, the process of difference application becomes clear: a model `M_1` and a difference model `Δ` are matched in the MMD_MM2MM transformation to obtain a new model `M_2`. `Δ` is calculated by computing the differences between two original models, `Δ = M_1' - M_1`. Model weaving leaves the door wide open to many possibilities, since `M_1` and `M_1'` don’t have to be the same models, they just need to conform to the same metamodel and can later be connected through a weaving model DAM:

- **Reconstructive application** is the standard way to weave them together. In this case `M_1 = M_1'`, and `Δ` matches completely against `M_1` without any further weaving. Hence `M_2 = M_2'`. This would be the normal way in software versioning. One model is stored and the other models can be obtained by the use of the differences, calculated out of exactly the same models.
- **Idempotent application** is the second and easy case: `M_1 ∩ M_1' = ∅` and the weaving model is empty. Thus the difference application will only add elements, described in MD. If there are no AddedMCs in MD, `M_1 = M_2`.
- **Transformative application**: if `M_1 ∩ M_1' ≠ ∅`, it is possible to realize a patch like application. The changes would only affect elements in the set `M_1 ∩ M_1'` and `M_2 ∩ M_2' ≠ ∅`. A special case of this would be `M_1' ⊆ M_1` with the result `M_2' ⊆ M_2`. Weaving techniques make it possible that there could also be matches even if `M_1 ∩ M_1' = ∅`, if elements from `M_1` and `M_1'` are connected in the weaving model. Transformative application can lead to several conflicts, for example, if `Δ` deletes an element, which exists in `M_1'` but not in `M_1`, so model weaving has to be done very carefully.
After describing the whole transformation process, let’s specify the MMD_MM2MM. This transformation is calculated by the HOT MMD2ATL, which gets the used difference metamodel MMD as input and outputs the corresponding MMD_MM2MM. A specific MMD_MM2MM transformation gets three inputs: an initial model, a difference model and a weaving model to connect them. For every class MC in the original metamodel, there are three rules in the MMD_MM2MM file (in this case, Δ means all differences, including those who are matched by model weaving):

- **AddedMC2MC**: creates for each AddedMC element in Δ a new instance of MC in M₂, according to the specifications of AddedMC.
- **ChangedMC2MC**: creates for each ChangedMC element in Δ a new instance of MC in M₂, according to the original specifications of MC in M₁ considering the changes which are done in ChangedMC and the associated updateElement.
- **UnchangedMC2MC**: copies all elements, which have to be the same in M₁ and M₂, hence all elements MC in M₁ which don’t have a related ChangedMC or DeletedMC in Δ.

It is easy to see, that all elements are able to be well dealt with: added elements because of the AddedMC2MC rule, changed elements because of the ChangedMC2MC. Deleted elements are not copied to the M₂ because of the restrictions in UnchangedMC2MC. Further information can be obtained in [3], section 5.2.

### 3.4 Operators

As mentioned before the evolution of software can consist of one complete version and differences to calculate the other versions. There are some techniques to ease the handling of those steps. First of all, we need a formalized definition of a delta:

\[
\Delta = (M₂ - M₁)
\]

This means, the difference between an initial model M₁ and a final model M₂ both conforming to the same metamodel MM, is stored in the delta Δ, conforming to a difference-metamodel derived from MM. The final model M₂ can be calculated by using M₁, Δ and a transformation T:

\[
M₂ = T_\Delta(M₁)
\]

The **Dual Calculation** makes it possible to automatically generate an inverse of a difference, making it possible to switch from forward to backward differencing and the other way
around or to simply undo last changes. Sequential Calculation and Parallel Calculation are two techniques, to merge several differences. Whilst the former uses linear grouping methods, the later merges two concurrent and competing differences on a horizontal level.

### 3.4.1 Dual Calculation

As mentioned before, we assume a transformation $T_\Delta$ between $M_1$ and $M_2$ with $\Delta$ representing the difference between $M_1$ and $M_2$. While $M_1$ and $M_2$ are models, conforming to the same metamodel $MM$, $T_\Delta$ conforms to an automatically generated difference metamodel $MMD$. Since we defined in 3.1 that a difference needs to be invertible, we induce the dual or inverse of the delta $\Delta$ as $\Delta^{-1}$. It is necessary to distinguish between the inverse of a difference representation and a inverse of a difference application. Whilst the former is our introduced inverse or dual representation of our difference, the latter is the undo-function, which can be formalized as $T^{-1}_\Delta$ with $M_1 = T^{-1}_\Delta(M_2)$. Since we define $\Delta^{-1}$ as the inverse of $\Delta$, we can assume, that $M_1 = T_{\Delta^{-1}}(T_\Delta(M_1))$ or in other words, the successive use of the difference and its dual nullifies the changes. So we can emulate the undo-function $T^{-1}$ by using the dual representation. Hence $T_{\Delta^{-1}}(x) = T^{-1}_\Delta(x)$. In this thesis, the undo operation is completely displaced by the transformation operation with the calculated dual of the last delta. This technique brings some benefit:

- The undo operation does not stick any more to a particular tool since every tool can implement the calculation of the dual from the difference.
- The undo operation does not need to store any more information since all needed information is stored in the delta itself.
- The dual difference matches the representation properties mentioned in 3.1 by construction; therefore it can be used in any way.

By looking closer to the dual calculation, it becomes obvious, why we want to record all and only the changes between the two compared versions. For example, it would be minimalistic to only store the deletion of a container and not the contained elements since the deletion of those could be automatically done. But it wouldn’t keep the self-contain or invertible conditions since the deletion of the contained elements is not contained in the delta, there wouldn’t be an adding of those elements in the dual delta. So those element would be missed in $M_1$ if we execute $M_1 = T_{\Delta^{-1}}(T_\Delta(M_1))$. It’s obvious, that every single element, which gets deleted or added, needs to be stored in the delta.

Relating to section 3.2, the dual calculation can be easily calculated: every AddedMC becomes a DeletedMC. Contrariwise, every DeletedMC becomes an AddedMC. The ChangedMC and the corresponding updateElement MC change position, or in other words: the ChangedMC is transformed into MC, the corresponding MC is transformed into ChangedMC and the updateElement relationship between them is reversed.

### 3.4.2 Sequential Calculation

There are possible circumstances where it is necessary and convenient, to group several deltas, which would be executed sequentially, into one simplified delta. The concept, calculation and furthermore optimization of this sequential delta will be extensively elaborated in chapter 4. At this point, it’s just necessary to introduce a sequential operator “;” which merges given $\Delta_1$ and $\Delta_2$ in the following way:

$$\Delta_{\text{merged}} = \Delta_1 ; \Delta_2.$$ 

Without loss of generality we can assume in the further discussion $m_1$, $m_2$, $m_3$ are models, which confirm to a metamodel $MM$. $\Delta_1$, $\Delta_2$, $\Delta_{\text{merged}}$ are deltas, which conform to a difference-
metamodel $MMD$, derived from MM. In this text, $\Delta_{\text{merged}}$ is also called the \textit{new delta}; $m_1$ is called the \textit{initial model}; $m_2$ is called the \textit{middle model} and $m_3$ is called the \textit{final model}. They are connected in following way:

- $\Delta_1 = (m_2 - m_1)$: the difference from $m_1$ to $m_2$
- $\Delta_2 = (m_3 - m_2)$: the difference from $m_2$ to $m_3$
- $\Delta_{\text{merged}}$: the optimal difference derived by merging $\Delta_1$ and $\Delta_2$

As the merged delta shall be optimal, $\Delta_{\text{merged}} = (m_3 - m_1)$ must be valid. Furthermore it is essential, that $T_{\Delta_1; \Delta_2} = T_{\Delta_1} * T_{\Delta_2}$ with \(*\) being the concatenation of two transformations. Referring to Figure 3.6 it is easy to see, since $T_{\Delta_2}(T_{\Delta_1}(m_1)) = m_3 = T_{\Delta_{\text{merged}}}(m_1)$.

3.4.3 Parallel Calculation

Often it is not possible to tell the order of two or more deltas which need to be applied. Hence, a sequential approach is not practical. For example, it’s quite common that several developers work on the same data or rather files, as mentioned in 2.4. Since locking of the file is impractical, it’s possible that two or more developers create a new version which leads to a new delta in each case. If those deltas are merged back to the repository, parallel calculation is used to get the actual result. Another example is the case when a branch is merged back into the trunk. In this case, sequential and parallel steps can even be combined, since first, the sequential steps in the branch are merged and later on, the final branch delta and the trunk delta can be merged in parallel. Similar to the sequential operator, a parallel operator does exist: $\Delta_{\text{merged}} = \Delta_1 \| \Delta_2$.

If the changes don’t interfere to each other, they are just merged (they could be treated as two sequential deltas where the order does not matter. Formalized this can be written as $\Delta_1 \| \Delta_2 = T_{\Delta_1; \Delta_2} + T_{\Delta_2; \Delta_1}$ with \(+\) being a non deterministic choice between the given merged deltas. If changes interfere, this solution is not practicable because syntactic and semantic conflicts may arise.

---

Figure 3.6: sequential composition
4 Sequential Calculation

Software evolution is one of the main targets for differences. As discussed in 3, there are several advantages in storing only one complete version plus the differences to calculate the other versions, for example the reduction of needed storage space. If the use of forward differencing is assumed (without loss of generality, see 3.4.1), it is necessary to consecutively apply every single difference in the version history to get the final version. To change from one version to another one - skipping some intermediate steps - it would be more favorable, to have an easy way to calculate one single difference, including all the needed information. Another example is the use of differences technology as software patches as mentioned in 3.3. If the actual deployed version is known, it is most easy to roll out a single difference, which represents the sequential calculation of all differences between the actual deployed and the actual developed version. The sequential calculation is also specified in the difference property **compositional** in 3.1.

For the sake of simplicity, we are assuming two deltas which are merged. It is easy to see that we can extend this approach to an arbitrary amount of deltas through concatenation. To prove this, we need the “;” sequential operator to be associative, hence $(\Delta_1;\Delta_2);\Delta_3 = \Delta_1;(\Delta_2;\Delta_3)$. It is easy to see, that this must be true because of minimalistic and self-contained properties, so a proof is set aside. Considering the associativity, we can prove the statement by mathematical induction:

- **Statement:** we can connect an arbitrary amount of deltas $\Delta_1 \ldots \Delta_n$ to a new delta $\Delta_{n\text{Merged}} = (\Delta_1 ; \ldots ; \Delta_n)$ by the sequential operator, which connects two deltas.
- **Basis:** $n=2$ ($n=0$ and $n=1$ are trivial): $\Delta_{2\text{Merged}} = (\Delta_1 ; \Delta_2)$ is given by the definition of “;”
- **Inductive step** ($n\rightarrow n+1$):
  \[\Delta_{n+1\text{Merged}} = (\Delta_1 ; \ldots ; \Delta_n ; \Delta_{n+1}) = (\Delta_1 ; \ldots ; \Delta_n) ; \Delta_{n+1} = \Delta_{n\text{Merged}} ; \Delta_{n+1}\]
- **Conclusion:** an arbitrary amount of deltas can be merged by merging two deltas to a new one and merge this new delta with the next delta and so on. q.e.d.

This approach is similar to the well-known divide-and-conquer algorithm[38]. The overall problem of merging several differences is divided into several small problems of merging two differences. This job can easily be “conquered” (solved) as later on shown.

4.1 Concept

For the sequential calculation of two deltas into a new delta there are only the source deltas $(\Delta_1, \Delta_2)$ and the conforming difference-metamodel MMD required. The calculation can be computed completely independent from any models or metamodels where $\Delta_1$, $\Delta_2$ and MMD were created by.

The Differences themselves can contain the original classes (MC) or their derived classes (addedMC, deletedMC and changedMC). Let’s call contained$(\Delta_j)$ the set of all original classes, which are either self contained in a delta $\Delta_j$ or represented by a derived class contained in $\Delta_j$. If an intersection of contained$(\Delta_1)$ and contained$(\Delta_2)$ equals the empty set, the deltas are sequential independent, formalized as contained$(\Delta_1) \cap$ contained$(\Delta_2) = \emptyset$. If
this is the case they can just be merged; hence all contained classes of $\Delta_1$ and $\Delta_2$ are copied to the new delta $\Delta_{\text{merged}}$.

If $\text{contained}(\Delta_1) \cap \text{contained}(\Delta_2) \neq \emptyset$, some optimization can and must be applied, regarding to the minimalistic property introduced in 3.1. Effectively, all information to calculate the intermediate steps needs to be deleted or rather not copied.

Class-based objects can belong to one of five forms in every delta: the MC itself, AddedMC, DeletedMC, ChangedMC or they do not appear.

If an object (in whatever form) does not appear in either $\Delta_1$ or $\Delta_2$, the object itself is sequentially independent and can be copied into $\Delta_{\text{merged}}$ in the form in which it exists in the other delta.

Deltas are a way to describe adding, deleting and changes. If an element appears in a delta in its original form MC, there are several reasons:

- It’s referenced by either a changed Element or an added element with one way association (by two way association, it would be of the type ChangedMC). So in fact, the element itself does not appear in the difference but only the reference to it. Hence, it can be treated as though it won’t exist in the delta.
- It is an updateElement of a ChangedMC. In this case, it’s included in the treatment of the ChangedMC element.

So finally there are the three derived forms (AddedMC, DeletedMC, ChangedMC) left, which leads us to 9 (= $3^2$) combinations, which are considered in section 4.2.

4.1.1 Constraints

Cicchetti mentions in section 4.2 of [3] that a calculated composite, like the sequential composite, needs to be valid data, thus conform to the same meta-model the composed documents conform to, which means $\Delta_1$, $\Delta_2$ and $\Delta_{\text{merged}}$ do need to conform to the same difference-metamodel. Also all properties introduced in 3.1 must be observed:

- **Minimalistic**: there are no elements added, which are not already elements of $\Delta_1$ and $\Delta_2$. Dispensable information, which can arise through the composition of two elements, is prevented by an optimized merging process;
- **Self-contained**, since all elements are copied and only the redundancies are reduced, all needed information is stored;
- **Transformative**: the used transformation can be the same as used for $\Delta_1$ or $\Delta_2$ since the deltas are based on the same difference-metamodel. Based on the fact, that $\Delta_1$ and $\Delta_2$ would leave models without elements specified in the difference models unchanged and the fact that no elements (in whatever form) are added, we can consider the transformative property to be upheld.
- **Invertible**: the optimization process makes sure, the $\Delta_{\text{merged}}$ is self-contained and minimalistic, making it possible to calculate the dual representation the same way it is calculated for $\Delta_1$ or $\Delta_2$.
- **Compositional**: A very important property since this is the basis for merging more then two deltas: $\Delta_{\text{merged}}$ itself is compositional and can be merged with every proper delta;
- **Model-based, meta-model independent, layout independent**, $\Delta_{\text{merged}}$ does only consist of elements which are found in $\Delta_1$ or $\Delta_2$ and uses the same representation technique, hence, these properties are fulfilled.
Considering the dual calculation \( T_{\Delta_1} \cdot (m_2) = m_1 \) and \( T_{\Delta_2} \cdot (m_3) = m_2 \) also some constraints over dual calculation can be done (remember \( \Delta_{\text{merged}} = (\Delta_1; \Delta_2) \) hence \( \Delta_{\text{merged}}^{-1} = (\Delta_1^{-1}; \Delta_2^{-1}) \))

- \( T_{\Delta_{\text{merged}}} \cdot (m_3) = m_1 \): the inverse of the sequential concatenation needs to transform the final model back into the initial one since the sequential concatenation itself transforms the initial model into the final one.
- \( (\Delta_1; \Delta_2)^{-1} = (\Delta_2^{-1}; \Delta_1^{-1}) \): thinking of the undo operation, it becomes obvious that to get back from \( m_3 \) to \( m_1 \) by single steps, first the changes done by \( \Delta_2 \) have to be reversed to get \( m_2 \) and afterwards the inverse of \( \Delta_1 \) needs to be applied to reach the initial model \( m_1 \). This means, that the inverse \( \Delta_{\text{merged}}^{-1} \) can be calculated by inverted sequential concatenation of the dual representation of \( \Delta_1 \) and \( \Delta_2 \), hence \( (\Delta_1; \Delta_2)^{-1} = \Delta_{\text{merged}}^{-1} = (\Delta_2^{-1}; \Delta_2^{-1}) \).

### 4.2 Cases

As mentioned in [4.1], there are nine cases obtained by combining all remaining classes with each other. They are composed in Table 4.1. The nine cases can be divided into two families: the existing and the non-existing family. Elements of the former family can appear. It is necessary to research in which way the according element in \( \Delta_{\text{merged}} \) can be calculated. All four elements of the latter can’t exist, represented in Table 4.1 by “–”. Nevertheless it is necessary to prove, that they can’t exist.

<table>
<thead>
<tr>
<th>( \Delta_1/\Delta_2 )</th>
<th>AddedMC</th>
<th>ChangedMC</th>
<th>DeletedMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddedMC</td>
<td>–</td>
<td>AddedMC</td>
<td>null</td>
</tr>
<tr>
<td>ChangedMC</td>
<td>–</td>
<td>null/ChangedMC</td>
<td>DeletedMC</td>
</tr>
<tr>
<td>DeletedMC</td>
<td>null/ChangedMC</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**Table 4.1: possible cases of sequential composition**

#### 4.2.1 Existing

As a side remark, ordered references are presented through an attribute `positionIndex` (compare chapter 4.4 of [3]). Hence they are covered by considering attributes and references in the following paragraphs.

- **AddedMC + ChangedMC**: an element is added in \( \Delta_1 \) and later on changed in \( \Delta_2 \). Since this element is added first, it did not exist in the initial model. Hence the new element in \( \Delta_{\text{merged}} \) needs to be of the type AddedMC, too. On the other hand, the changes in the ChangedMC must be paid attention: all attributes and references which are changed in the ChangedMC must be changed in the new AddedMC compared to the original AddedMC. All other parts of the original AddedMC can just be copied to the new one.

- **ChangedMC + DeletedMC**: inversely to AddedMC + ChangedMC element is deleted in \( \Delta_2 \), so it does not appear in the final model. Hence the new element in \( \Delta_{\text{merged}} \) needs to be of the type DeletedMC, too. Since the element, which is deleted in \( \Delta_2 \) differs from the element which needs to be deleted in \( \Delta_{\text{merged}} \), all changes in attributes and references which appear in ChangedMC must be treated in the new DeletedMC.

- **AddedMC + DeletedMC**: the element is added in \( \Delta_1 \), so it did not exist in the initial model. It is deleted in \( \Delta_2 \), so it does also not exist in the final model. Hence, it mustn’t appear in the \( \Delta_{\text{merged}} \).

- **DeletedMC + AddedMC**: if the element, which is added in \( \Delta_2 \), has exactly the same attributes and references then the one which is deleted in \( \Delta_1 \), there is no change in the models and the element mustn’t appear in the \( \Delta_{\text{merged}} \). If there is a change,
4.2.2 Non Existing

DeletedMC is copied to $\Delta_{merged}$ as new ChangedMC and AddedMC is copied to $\Delta_{merged}$ as the corresponding updateElement MC. Attributes or references which do not differ mustn’t be copied.

- **ChangedMC + ChangedMC**: if ChangedMC in $\Delta_2$ nullifies the changes made by the ChangedMC in $\Delta_1$, the element mustn’t appear in the new $\Delta_{merged}$. In other words, if the calculated dual or reverse of the one ChangedMC equals the other ChangedMC, both can be ignored. If there are differences, a new ChangedMC need to be constructed in $\Delta_{merged}$: Let’s assume $chg_x$ are the set of attributes and references which appear in the ChangedMC in $\Delta_x$ and $upd_x$ are the attributes and references that appear in the updatedElement MC in $\Delta_x$. Consider that $chg_x-upd_y$ means all elements which appear in $chg_x$ but not in $upd_y$. Two construction rules can be defined (see Figure 4.1)

  - $chg_{merged} = (chg_1-upd_2) + (chg_2-upd_1)$.
  - $upd_{merged} = (upd_1-chg_2) + (upd_2-chg_1)$.

![](image)

**Figure 4.1: concatenation of two changed classes**

4.2.2 Non Existing

- **AddedMC+AddedMC**: it is impossible to add an element which already exists in a model. Since the element is added in $\Delta_1$, it exists in $m_2$. Hence it can’t be added in $\Delta_2$ according to the definition, $\Delta_2$ is the difference from $m_2$ to $m_3$.

- **ChangedMC+AddedMC**: After the change of an element, the element does still exist (unlike when it is deleted). So the element still exists in $m_2$ and hence can’t be added in $\Delta_2$.

- **DeletedMC+ChangedMC**: the change operation in $\Delta_2$ needs an existing element in $m_2$. Since the element was deleted in $\Delta_1$ it does not exist anymore in $m_2$.

- **DeletedMC+DeletedMC**: to delete an element in $\Delta_2$, the element must exist in $m_2$. Since it is deleted in $\Delta_1$, it doesn’t exist in $m_2$.

4.2.3 Examples

This chapter presents some examples to the sequential composition cases. For Figure 4.2 and Figure 4.3 an empty initial model and for Figure 4.4 and Figure 4.5 an initial model with only a class $c1$ (name="class1", isAbstract="false") can be assumed.

**Figure 4.2** shows the **AddedMC + ChangedMC** case. In $\Delta_1$ a class $c1$ is added with the attributes name = class1 and isAbstract = false. This attribute is changed in $\Delta_2$ to class1b. In $\Delta_{merged}$ those two differences are merged. Since there was no class $c1$ in the initial model and there is one in the final model, a class needs to be added in $\Delta_{merged}$. This class has the isAbstract value of the AddedClass in $\Delta_1$, but since the name was changed in $\Delta_2$ it features the new value class1b.

**Figure 4.3** shows the case, where a class is added in $\Delta_1$ and exactly the same class is deleted afterwards in $\Delta_2$, hence this class does not need to appear in $\Delta_{merged}$ (it can’t be deleted or
changed since it does not exist in the initial model and it can’t be added since it does not exist in the final model).

Figure 4.4 shows two merged changes. The Change in the $\Delta_{\text{merged}}$ contains changed elements from $\Delta_1$ (the name attribute) and from $\Delta_2$ (the isAbstract attribute).

Figure 4.5 finally gives an example to the fact, that sometimes the optimization can be very simple and sometimes not: if the AddedClass in $\Delta_{2a}$ has exactly the same attributes and references as the DeletedClass in $\Delta_{1a}$, the class $c1$ in the initial model equals the class $c1$ in the final model. Hence, there’s no difference needed and $\Delta_{\text{merged}_a}$ does not contain information to $c1$. If the classes differ like between $\Delta_{1b}$ and $\Delta_{2b}$, instead of no information, a change operation is needed, visualized in $\Delta_{\text{merged}_b}$.
4.3 Implementation

The Implementation is done in several files. To merge two deltas, an ATL Transformation is used which gets two deltas as an input and gives the merged delta as the output as seen in Figure 4.6. Since an ATL transformation needs the corresponding metamodel (in this case MMD) as an input information, it needs to be calculated by a HOT MMD2seqATL. So finally, there are two transformations: MMD2seqATL is a HOT, which constructs a new MD_MD2MD ATL file (for naming replace the “MD”’s in the name with the corresponding MMD name) based on either MM or MMD (if MMD is used, there’s calculation needed to calculate MMD from MM, on the other hand, since we work with deltas this calculation needs to be done anyway and if MMD is used there’s no need for MM in the complete calculation. So in my opinion, MMD is preferable). In the following

MMD2seqATL.atl
First of all, the transformation, which merges two deltas, needs to be calculated based on the used MMD. This is done by the mentioned HOT MMD2seqATL.atl. It gets a KM3 based metamodel as input and outputs the MD_MD2MD transformation based on the input metamodel. For example, with the input simpleUMLdiff.km3, the HOT would construct the simpleUMLdiff_simpleUMLdiff2simpleUMLdiff transformation, which can merge all differences based on the simpleUMLdiff MMD. The MMD2seqATL.atl constructs the following parts:
The header of the new Transformation; five rules (later described) for every non abstract class based element which is no Added, Changed or Deleted element, to handle the existing cases; several corresponding helpers.

MD_MD2MD.atl
The constructed MD_MD2MD gets $\Delta_1$ and $\Delta_2$ as input and gives $\Delta_{\text{merged}}$ as output. Let’s assume there’s only one class MC in MM. So we need to have the following matched rules:
- AddedMC_ChangedMC2AddedMC
- ChangedMC_DeletedMC2DeletedMC
- DeletedMC_AddedMC2ChangedMC
- ChangedMC_ChangedMC2ChangedMC
• no_siblingMC2copy: this rule is necessary to copy all elements, which are mentioned in one of the deltas, but not in the other one.
• AddedMC_DeletedMC2null: in this case, there’s no need for a new element in $\Delta_{merged}$ so, this rule can also be omitted and will be in the implementation.

The AddedMC_ChangedMC2AddedMC handles the case that an element is added in the first difference and is changed in the second one. It checks for all structural features (attributes and references) of AddedMC, if they were changed. If so, the new AddedMC gets the changed structural features otherwise the original added. This check is done by a helper, called getMergedMC[name of feature]. The ChangedMC_DeletedMC2DeletedMC works pretty much the same way.

The next two cases become a little more delicate. The two differences can eliminate each other (if the second difference undoes exactly what the first difference changed). So in the input section needs to be checked, that this is not the case (hence, the there’s no differences created in the merged delta). If there is a change, a new empty ChangedMC class with corresponding updateElement is created. In the do section of the rule, the structural features, were a change occurs are added to the ChangedMC and the updateElement respectively. In the DeletedMC_AddedMC2ChangedMC it is checked, if a structural feature, which was deleted, is added in the original version. If not so, the version of the DeletedMC is added to the ChangedMC whilst the version of the AddedMC becomes part of the updateElement. In the ChangedMC_ChangedMC2ChangedMC the structural feature of the first ChangedMC and the feature of the updateElement of the second ChangedMC are compared and copied to the new ChangedMC and updateElement respectively if they diver. The check, if the two differences eliminate each other is done with the helpers DeletedAddedMCEliminated and ChangedChangedMCEliminated. A further helper changedClassUpdExists checkes, if both ChangedMC have an updateElement.

Last but not least the no_siblingMC2copy copies exactly all those elements, which are in one difference but not in the other. Since ATL forbids a rule with no output, the rule AddedMC_DeletedMC2null is obsolete. In fact it is also unnecessary because we don’t use the ATL refine mode (compare 2.2.1).

The automatically launching and execution of the described ATL files can be done through ANT-scripts, which are not a part of this thesis. Just to mention, since ANT files are xml-based, they can be also an output of an ATL Transformation. This could be an approach to compose more then two deltas. The transformation could get a list of differences as input and create ANT scripts as output, which run the MMD2seqATL.atl and later on use the created MD_MD2MD.atl to compose all differences as described in the preamble of chapter 4. In the following sections 4.3.1 and 4.3.2 two problems of the shown methods are described. Section 4.3.3 touches an idea how to test the result of a merged delta.

4.3.1 First Problem: adding or deleting of contained elements

Through the research of the implementation, an interesting problem occurred. As you can see in figure 6.5 of [3], in the original approach deleting of an element contained by another element is done by changing the original element, having the element, which shall be deleted, associated with the ChangedMC. Since it is not associated with the belonging updateElement MC, it is known to be deleted. The other way round, if an element shall be added to a container, which already exists, it is done by associating the MC (which shall be added) to the updateElement of the MC of the container where it shall be added.
This leads to some trouble at the optimization of differences as seen in Figure 4.8. Consider an empty initial model $m_1$. In the first step $\Delta_1$ a class $c_1$ is added. In the second step $\Delta_2$ an operation $o_1$ is added to this class by changing the class $c_1$. So in the final model, there is a class $c_1$ with a contained operation $o_1$. Easy to see, that a concatenation of those two deltas needs to lead to a delta $\Delta_{\text{merged}}$ which include an AddedClass (for the $c_1$ class) connected through a container reference with an AddedOperation (for the $o_1$ operation). The calculation of AddedClass is covered by the AddedMC + ChangedMC = AddedMC case. The change from \texttt{"Operation"} to \texttt{\"AddedOperation\"} is covered by no rule. This case is an example which makes it necessary to research the references of an element to find out whether it needs to be adapted through a rule or not. A Solution to this problem is presented in chapter 5.1.

4.3.2 Second Problem: changing an contained element

If a contained element is changed, the container is changed to as seen in figure 4.9 of [3]. This handling of containers also tends to result in trouble. Compare two changes case in chapter 4.2.1 if two changes nullify each other, the element can be ignored. It is quite hard to detect whether an element is just changed because a contained element is changed or if there are no changes because they nullify each other. To anticipate: this problem can be solved by only use the \texttt{"changedMC"} class if an element is actually changed (hence if an attribute or a reference of it is changed). A basis for this approach is presented in chapter 5.1.

4.3.3 Testing

It is quite easy to check, if an optimal delta was calculated correct: based on the fact that the calculated $\Delta_{\text{merged}} = (m_3 - m_1)$ is the difference between the final and the initial model. Since
the final model is also reachable through incremental usage of the single differences $\Delta_{\text{merged}}$ is based on, it is possible to use this fact to calculate $m_3$ in two different ways: first by using our calculated $\Delta_{\text{merged}}$. Second by applying all in-between steps (in this case $\Delta_1$ to reach $m_2$ and $\Delta_2$ to finally reach the final model $m_3$; hence $T_{\Delta_2}(T_{\Delta_1}(m_1)) = m_3$). We can use the difference calculation algorithm to gain the difference $\Delta_{\text{allSteps}}$ between the initial model $m_1$ and the final model $m_3$ derived by applying all single difference steps, which where earlier used to calculate $\Delta_{\text{merged}}$. Considering all constraints, this difference $\Delta_{\text{allSteps}}$ must equal our calculated optimal difference $\Delta_{\text{merged}}$. 
5 Personal Conclusion

Since I heard about model-driven-engineering the first time in university, I did know that this is the next big thing. Raising the level of abstraction did always improve quality and lower the costs. As a friend of software design, I also like the idea of separating the designing process from the coding. Keeping track on the developments, I was always happy about new techniques which ease the exchange and porting of software, data and information, since the different approaches, syntaxes etc of various research institutions were an annoyance in my opinion. MOF, KM3, Transformations and HOT but also the use of models itself are a great opportunity, to solve the problem of differing formats. Differences itself are a great way to handle the problem of storing information in different versions. The approach of metamodel independent differences is brilliant in my eyes. It is simple and powerful at the same time.

My announced research to the topic of sequential compositions is only adding a little patch to this work. Still, I think, that it is an important part, since it adds the possibility of calculating with differences, which will ease the handling of software evolution. Finally, a problem appeared while I tried to do the implementation part, resulting in a very complicated work. By researching this phenomenon, I discovered a simple solution for it, which – in my eyes – goes perfectly with Cicchetti, Di Ruscio, and Pierantoniaio approach.

5.1 Changing the Problem

In section a problem was presented, which brings heavy complexity into the sequential calculation. By looking at Figure 4.8 the problem and the solution becomes clear: either in $\Delta_{\text{merged}}$ the operation $o_1$ needs to be added as “Operation”, too. Or it needs to be added as an AddedOperation in $\Delta_2$, too. This leads directly to a more broad view of the whole concept and brings up a fundamental question: are contained elements parts of the container or do they need to be treated as single entities?

![](image)

**Figure 5.1: new approach of deleting an adding of contained elements**

Based on the simpleUML example, it would make sense to treat contained elements like parts of the container. Just think of a class, which contains methods and attributes. They can’t exist without a class; hence they can’t be added or deleted without a containing class. On the other hand this case, it would be needless to have addedMC or deletedMC for contained classes. Also, thinking of these methods and attributes, you think of “adding” attributes and methods to a class, so that the class gets changed. You don’t think of changing the class with the result that some new methods or classes appear. Treating contained
elements as part of Cicchettis metamodel independent approach makes it necessary to also treat contained elements in a coherent and consequential way. I highly recommend to always use AddedMC and DeletedMC, regardless if MC is a contained element or not, as visualized in Figure 5.1. By the way: the containing class, to which elements are added or deleted, needs still to be changed, because of the changing references which need to be added or deleted in this class. Treating contained elements as single entities also solves the problem presented in 4.3.2: as a single entity, changing of an element doesn’t interfere with the containing class, as long as the reference to this class isn’t changed.

There’s a simple proof for this solution, based on the famous families2persons example, or rather on the families metamodel:

Figure 5.2: two versions of a family

Consider the following situation. A model $m_1$ consists of a family “Max” (for the sake of simplicity family members which do not occur in this example were set aside). In a second version, a son “Wilhelm” is added (born) to this family, which directly becomes the father of a second family “Moritz” (which is obviously added to the model, too).

With the old approach, it would be impossible to create a difference between the two models, without loss of the minimalistic property, since it would look like in Figure 5.3. The element “p” would be in the difference as Member and as AddedMember.

Figure 5.3: $\Delta_1$ in the old approach

Using the new approach, it is possible, to use a single “AddedMember” entity, to cover both cases as seen in Figure 5.4.

Figure 5.4: $\Delta_1$ in the new approach
5.2 Future Work

As mentioned at the beginning, MDE is a rich field of research. The metamodel independent approach announced by Cicchetti, Di Ruscio and Pierantonio is a forward-looking approach. There’s already research ongoing for example in

- **Parallel Composition**: as described in 3.4.3, this topic is more complicated than sequential compositions. Several conflicts can appear, necessitating mechanisms to store and solve them.

- **Deltas as Patches**: applying patches to differing models makes it necessary to solve the identifying of elements. Model weaving is a good technique to solve this problem.

Also, I highly suggest collecting, formalizing and publishing a list of rules how differences are constructed. This could ease and disambiguate future research. Those rules could look like:

- If a container is deleted, all contained elements which are not contained in another container, need to be deleted

- Elements of class MC can only be deleted through a DeletedMC

- If an element x, which is referenced by another element y, is deleted, element y needs to be changed.
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>ATL Development Tools</td>
</tr>
<tr>
<td>AMMA</td>
<td>ATLAS Modell Management Architektur</td>
</tr>
<tr>
<td>AMW</td>
<td>ATLAS Model Weaver (see 2.3.1)</td>
</tr>
<tr>
<td>ATL</td>
<td>ATLAS Transformation Language (see 2.2.1)</td>
</tr>
<tr>
<td>CIM</td>
<td>Computation Independent (see 2.1.2)</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System (see 2.4)</td>
</tr>
<tr>
<td>DSM</td>
<td>Domain-Specific-Modeling (see 2.1.1)</td>
</tr>
<tr>
<td>HLPN</td>
<td>High-Level Petri Nets</td>
</tr>
<tr>
<td>HOT</td>
<td>Higher Order Transformation (see 2.2.2)</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>INRIA</td>
<td>Institut national de recherche en informatique et en automatique</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
</tr>
<tr>
<td>KM3</td>
<td>Kernel MetaMetaModel (see 2.1.4)</td>
</tr>
<tr>
<td>M0-M3</td>
<td>Level 0-3 on the four-level architecture (see 2.1.3)</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture (see 2.1.2)</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering (see 2.1)</td>
</tr>
<tr>
<td>MMD</td>
<td>Difference MetaMetaModel</td>
</tr>
<tr>
<td>MM</td>
<td>MetaModel</td>
</tr>
<tr>
<td>MMM</td>
<td>MetaMetaModel</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility (see 2.1.3)</td>
</tr>
<tr>
<td>MOF/QVT</td>
<td>Meta-Object Facility/Query, View, Transformation</td>
</tr>
<tr>
<td>MTL</td>
<td>Model Transformation Language (see 2.2)</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model (see 2.1.2)</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model (see 2.1.2)</td>
</tr>
<tr>
<td>RVS</td>
<td>Revision Control System (see 2.4)</td>
</tr>
<tr>
<td>SVN</td>
<td>Subversion (see 2.4)</td>
</tr>
<tr>
<td>VSS</td>
<td>Visual SourceSafe (see 2.4)</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Reference

11. Jean Bézivin, Frédéric Jouault, David Touzet, 2005. *An introduction to the ATLAS Model Management Architecture*


32. Cambridge Dictionaries Online, Definition of differ, difference, different [Online] Available at: http://dictionary.cambridge.org/define.asp?key=21636&dict=CALD


39. ATLAS group LINA & INRIA Nantes, 2005. ATL Starter’s Guide (version 0.1)