Rijksuniversiteit Groningen
Instituut voor Wiskunde en Informatica

Tampere University of Technology
Department of Information Technology

Jan Salvador van der Ven

An Implementation of Set Operations on UML Diagrams

Master of Science Thesis

Supervisors: Prof. dr. ir. J. Bosch, Rijksuniversiteit Groningen
Prof. dr. W.H. Hesselink, Rijksuniversiteit Groningen
Prof. K. Koskimies, Tampere University of Technology

Abstract

Models are used intensively during the design, implementation and evolution of software systems. Because the software systems are getting larger, models are getting larger too. Consequently, there is a demand for automated tool support to process these models. The basic set operations, applied on models, provide a way of comparing and merging models. This thesis discusses an implementation of set operations on UML class diagrams. They are implemented in a CASE-tool independent environment. The set operations are used to merge, slice and check UML models and the implementation is tested on big industrial forward- and reverse-engineered models.

Table of Contents

1 Introduction

Within software engineering modeling is taking a more and more primary position. Currently, the Unified Modeling Language (UML) is the most important modeling language in the field. Many modeling techniques, methods and CASE tools use UML. There is a demanding need for automated processing of UML models. During the reverse-engineering and software maintenance process huge models have to be compared. The MDA initiative leans heavily on the existence of model processing operations. As product lines are designed, different versions of models need to be constructed, compared and sliced.

Many CASE tools support UML, but only with partial model processing. There is a need to fill the gap which exists in the tool support and the demands of the designers. The basic set operations: union, intersection and difference can be used to process models. I will present an implementation of these set operations on UML Class diagrams. The implementation is founded on the work of Selonen [Sel03], but it presents a concrete implementation of the set operations on UML Class diagrams while Selonen specifies the set operations at very abstract level. The abstract criteria from Selonen [Sel04] are used to create criteria for the correspondence detection of Class diagrams. Various ways of presenting the results have been developed, like coloring and using statistical data. This implementation is created on a CASE-tool independent model processing platform, and is highly adjustable and potentially usable for other diagram types. The implemented set operations have been used on real life forward- and reverse-engineered industrial models and have proved to be useful to merge, slice and check big models.

The next chapter gives a short introduction about the idea of modeling, modeling in computer science and UML. In the Problem Statement, Chapter 3, the demand for model processing and what the set operations can do for that is discussed. At the end of the Problem Statement a few scenarios as well as the main thesis questions are stated. Then the analysis of the problem and the implementation issues are discussed in Chapter 4 and 5. In the Validation Chapter the scenarios from the Problem Statement are analyzed by using test cases on real life models, followed by a discussion of the thesis questions. The last Chapter discusses some concluding remarks and suggestions for further development of the set operations.

2 Background

2.1 Modeling

In this thesis the concepts model and modeling are often used. But what is modeling? What is the connection between a model and the real world? Every science and engineering field uses models intensively. They come in different shapes and different forms. Some examples of models are models of: buildings, cities, boats, economic development, business processes, the earth (maps), the universe, weather, computer data, computer programs, etc. These models show a simplification, abstraction or a building-plan of something.

What is the use of constructing models? Do they tell us more about the object we are modeling? Models make it easier to learn things about reality, by talking about the models instead of the reality itself. The main purpose of using models is for *communication* about the modeled objects and for *abstraction* of these objects. To discuss about the design or properties of an object, it is necessary to have a view (or more views) which all the stakeholders agree on. Especially when the modeled object is hard to comprehend by itself, it is easier to use an abstracted model.

The model is not intended for modeling reality as such. [Kent78]

Is a model sufficient for describing a house, a boat or a computer program? Or perhaps, is the model the house, the boat, the computer program? For the housing example it is not really sensible to say that the model is the house. But within the computer science the gap is getting smaller. With several commonly used CASE tools it is already possible to generate the basic source code from the model, and visa-versa: to generate models from the source code. Enhancing the status of models to first-class citizens of the design process is one of the goals of Model Driven Architecture (MDA) as described in [OMG03b]. In the future complete model-based design will be possible, where the model can be executed directly. An example in this direction is xUML, designed by [Ken03].

Assuming that the model is not yet considered to be the object itself, three aspects have to be taken into consideration when constructing models.

- *Context.* What is the model describing? A boat, a house, a computer program?
- *Language and naming.* Are the terms used ambiguously? Do all the users of the model agree on the used terms?
- *Knowledge of user.* Are the users skilled enough to interpret the model correctly?

Many books have been written about modeling. Kent [Kent78] presents a philosophical analysis of the nature of data and models. The examples used are sometimes a bit old, but the general idea is still quite up-to-date. A more complete view on modeling is given in [Bom97]. Both books focus on the modeling of computer systems, but contain a good introduction to general modeling too.

In this thesis the focus will be on the modeling of computer systems. These are constructed before the system is created, as well as (ideally) constantly updated during the system development and maintenance. They are used to gain a common understanding of the system for the different designers, architects, managers, and other stakeholders who work with the system. Some aspects of computer modeling, as well as the leading modeling language (UML) is discussed in the next section.

2.2 Modeling in computer science

2.2.1 Pre-UML

Once upon the time, computers and computer programs were simple. Programs could be constructed by one person and he could explain the functionality of a program to someone else without big problems. But when the complexity of computers increased, problems arose.

The first problems emerged when the amount of data increased. Models were needed to arrange the data in portions which could be comprehended by the developers. Usually the programs were still simple enough to understand and explain without complex modeling. An example of data modeling which is still being used is database design.

When the complexity of computers kept increasing, there was also a need for modeling computer programs, especially when programs got so big that more than one designer was needed to construct it. During the late 1960s, software engineering was born, aiming to solve the problems of software development in a systematic way. With that, the development of modeling techniques for computer systems speeded up.

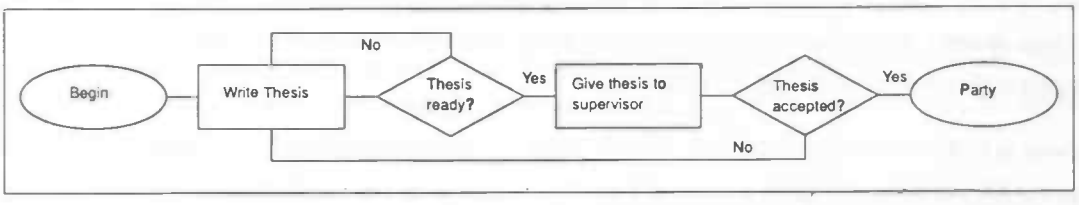

Figure 2.1: A flowchart diagram

At first most of the models were constructed with an algorithmic approach. This approach was procedure and function based. A typical example of algorithmic models is a flowchart diagram, as shown in Figure 2.1. After the introduction of object oriented programming languages in the mid 70s, new modeling techniques were developed.

2.2.2 The birth of UML

When usable programming languages for object oriented programming were born (Smalltalk and C++), also methods for object oriented modeling were needed. In the 80s and 90s many different methods and models for object oriented design where developed. There were many different modeling paradigms and methods, which had different expression powers and different strengths. Most of the models had some things in common, like a same type of diagram, but they were used slightly differently. Many designers favored one method over another and some fights were fought over it (the 'method wars'). This called for some sort of standardization.

The standardization started when Grady Booch and Jim Rumbaugh began working on unifying their methods, the OMT [Rum91] and Booch [Boo91] methods, in '94. This resulted in the presentation of the draft of their first combined method in '95; the Unified Method. In this same period Ivar Jacobsen joined them at Rational Software, contributing the concept of use case diagrams [Jac91]. Throughout the next year these three amigos started working on the Unified Modeling Language (UML).

In the following years more companies invested in the development of UML, which resulted in UML version 1.0. The Object Management Group (OMG) was looking for a serious standardized model, and asked for proposals. UML version 1.0 was submitted to them in January 1997. They adopted UML and from then on it was promoted as the standard modeling language. currently UML is being developed completely by OMG. Version 1.5 [OMG03] is used widely nowadays, while version 2.0 is in its finalization phase.

2.2.3 UML

With the increase of the complexity and the size of computer systems, the field of software engineering started to develop. Software engineering was meant as a systematic way to analyze, design and implement software, seen from a typically practical point of view. The most important software attributes at this time, according to [Som98], are maintainability, dependability, efficiency, and usability.

Modeling techniques are used to construct software systems which possess these attributes at the highest level. by using models it is easier to predict the resulting attributes before a system is developed. For different software systems, with different demands, different kinds of models are used. UML tries to be the general solution to the modeling problems and differences.

So what is UML? According to [OMG03, p. xxv], "UML is a graphical language for visualizing, specifying, construction and documenting the artifacts of a software-intensive system". UML specification describes the syntax (and some semantics) of the model structure. In this it is using terms which could be unambiguous. A description of the abstract syntax is given, with guidelines for well-formedness and notation, as well as some semantics.

UML is a modeling language. This means that it does not specify a construction method or a development process, but just a way of writing down the ideas in a particular way. Although Jacobson has created the

Unified Process [Jac99], a software development process which uses UML, it is not necessary to use this process when using UML. Actually most of the software development processes use UML these days.

2.2.4 Post UML?

In the software engineering community UML is used intensively. No real big competitors with such an expressive power as UML are at hand. In specific fields of software engineering other models are still used, but they tend to move toward UML. Because UML gives the designer the possibility to define parts of it self, other modeling languages can be made 'part of' UML. An example of this is the Specification and Description Language (SDL), which is still used separate from UML, but with the extension mechanisms of UML it can be added, as described in [Bjö00].

It is hard to look into the future, but as long as most of the biggest software companies keep using UML, it will probably stay as the leading modeling language. Because UML keeps evolving, it will most likely be able to adapt to future demands for modeling. After all, as modeling techniques are to be used to communicate, it is quite easy when all the modelers speak the same language, even though they might use different subsets or versions if it.

As a result of the more intensive tool support and the success stories of model based projects, there is an increasing demand for making models the first-class citizens of software design. In the initiative of Model Driven Architecture [OMG03b], various UML models are used to describe computer systems. In MDA the Platform Independent Model (PIM) can have different Platform Specific Models (PSM). These PSM will be implementations of the system in the near future, according to some MDA-fanatics. Automated processing of models is essential to the success of MDA.

2.3 UML

2.3.1 Different types of diagrams

UML offers nine different types of diagrams. They all contain a different view on the modeled system. These diagrams are discussed briefly below. A wider introduction to UML is given by Fowler [Fow00] and Booch [Boo98]. The different types of diagrams are:

- The *use case diagram* captures the requirements and shows the interaction from the user (either human or other system) to the system.
- The *class diagram* describes classes of a system and the connections they have. Class diagrams from different levels of abstraction can be used in the same model.
- The *object diagram* shows a possible 'snapshot' of the system at a certain time, as an instance of a class diagram.

- Interaction diagrams describe the behavior of the system, typically in a certain use case. There are two types of interaction diagrams: the *sequence diagram* and the *collaboration diagram*.
- The *state diagram* shows all the possible states a certain class can have, as well as the transitions between these states.
- The *activity diagram* visualizes the order in which the activities of a state take place.
- Physical diagrams show the bigger pieces of design. The *deployment diagram* describes the connections between the used hardware, and the *component diagram* shows the dependencies between the components of the system.

Note that most of the diagrams described above can be used in more than one way, at different places in the development process and at different levels of abstraction. By using different (UML-specified) techniques like stereotypes, the models can be enriched and made more suitable for the specific developed system. Many companies have developed their own dialect of UML.

Because the rest of this thesis will discuss the set operations on class diagrams, they are discussed here in somewhat more detail. Class diagrams are used to show the different objects in the system and their relations. They are used in most of the object oriented design methods. They typically contain information about the classifiers, e.g. attributes and operations, as well as the relationships between classifiers. These classifiers can be either classes or interfaces. There are three types of relationships: association, generalization and dependency. These relationships can be further stereotyped.

Class diagrams can be used at different abstraction levels of the modeled system. The diagram can be drawn to represent the system on implementation level, where the classes in the diagram correspond with the objects in the implementation. It can also be used to give a general overview of the whole system, without connecting the classes to direct objects in the implementation. These perspectives are shortly discussed in [Fow00].

2.3.2 UML Metamodel structure

UML is described as a Meta Object Facility (MOF) [OMG02] based metamodel-description. The model itself is a metamodel instance of the MOF system. In the description of UML as well as MOF, the notations of UML class diagrams are used. In [OMG03] UML is described in a four layer metamodel architecture: the meta-metamodel layer, the metamodel layer, the model layer and the user objects layer. The four layers used are shown in table 2.1, which is taken from the UML specification.

As can be seen from table 2.1, all the layers are instances of the layer above (except for the meta-metamodel layer which is self-describing). So a metamodel is an instance of a meta-metamodel, a model is an instance of a metamodel and a user object is an instance of a model. UML is defined at metamodel level. So this is an instance of the meta-metamodel level, for which MOF is used, as defined in [OMG02].

As an example of what the metamodel level means consider figure 2.2. It describes a subset of the metamodel for structural diagrams. The metamodel is used in the problem analysis (Chapter 4), where the set operations

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	Class, Attribute, Operation, Component
model	An instance of a metamodel. Defines a language to describe an information domain.	StockShare, askPrice, sellLimitOrder, StockQuoteServer
user objects (user data)	An instance of a model. Defines a specific information domain.	<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>

Table 2.1: The UML metamodel structure

are discussed at metamodel-level. The metamodel is defined as a class diagram, containing Classes which can contain Attributes, and Generalizations and Associations between Classes. A special type of Association, the Composition, is used to represent the whole/part relationship between the Classes, it is represented as a black square at the end of the Association.

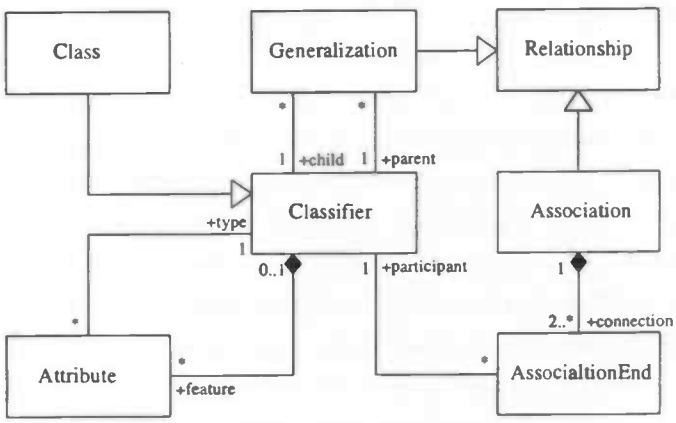

Figure 2.2: An example part of the UML metamodel

The UML metamodel is not a system design level model. It is a logical model, which can be used to construct a physical or implementation model. Together with the diagrams every metamodel element is described in [OMG03] in natural language as well as in the Object Constraint Language (OCL). For most of the users of UML there is no need to know about the metamodel layering structure. When designing models, the work is done at model-level.

2.3.3 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is part of the UML standard, as described in [OMG03, p.6-1]. It is used to create expressions and constraints on UML, but it is not restricted to UML. Simple logic with Boolean expressions like AND, OR, NOT and IMPLIES enriched with predicate logic statements like ForAll and Exists are used. It is applied within the UML standard to specify constraints for using the models (the well-formedness rules). An example of an OCL statement, taken from the UML standard [OMG03, p.2-61] is shown in the next figure.

```
-- OCL example

self.allContents -> select(oclIsKindOf(Association))->
    forAll(a1, a2 |
        a1.name = a2.name and
        a1.connection.participant = a2.connection.participant
        implies a1 = a2)
```

Figure 2.4: An example of OCL(taken from [OMG03, p.2-61])

This example shows that all Associations must have a unique combination of name and associated Classifiers in the Namespace. OCL is used in Chapter 4 to describe the constraints for correspondence. Minor knowledge of OCL is assumed there.

3 Problem Statement

3.1 Model processing

Software sensitive systems evolve during their life-time. With the introduction of new requirements, direct or indirect, the system needs to adopt. Especially with product line design the control of the evolution and versioning of the system is critical [Bos00]. By using models throughout the development and evolution of the system the change can be controlled and monitored. Models are currently getting more expression power and research nowadays is highly focused on putting them as first-class-citizens of the design and evolution of software systems. Therefore, there is a demanding need for good model oriented tool support. These tools should be able to:

- *Merge* models: to combine models or model fragments into one model. This is for example useful when the different model fragments have been developed by different developers. Or, in a situation that model fragments represent specific functionality; these fragments can be merged to create the model which has all of the desired functionality. This is very useful in product line design as well as in the MDA initiative.
- *Check* models: to check models for consistency, differences, similarities, etc. When model fragments come from different types of diagrams they can be checked against each other for inconsistencies. Also models can be checked against their re-engineered counterparts, to verify whether the model is implemented correctly. A third use of checking is when different versions of models are available. Seeing the changes could give a better insight in the design process, or find errors or violations.
- *Slice* models: to cut models, or model fragments, into interesting slices. This is for example useful to show the impact of a feature (represented as a model fragment), on the whole model. When combined with transformations, model fragments from other types of diagrams can be used. So for example the influence of a usage scenario can be visualized in a class diagram.
- *Synthesize* models: to produce models automatically from other types of models. This is a direct transformation which is useful to obtain information from other diagrams.

Because none of the current CASE tools gives good support for these operations, the Institute of Software Systems of the Technical University of Tampere (TUT) is developing a platform for UML model processing [PRA03].

3.2 Set Operations on UML diagrams

Two types of operations are essential for the support of the merging, checking, slicing, and syntheses of models. The first one is a *transformation* from one UML diagram to another. This is a unary operation $D \rightarrow D$, where D represents a diagram and the resulting diagram is of a different type than the original one. A transformation is not always useful, or even possible, but in some cases it can be very valuable. In [Sel03b] the possibilities for this type of operation are discussed.

The second type of operation is the *set operation*. These are defined in [Sel03] as binary operations $D \times D \rightarrow D$, who produce one UML diagram out of two input diagrams. The used diagrams are of the same diagram type. There are three basic set operations: union, intersection and difference. The set operations are the most natural on structural diagrams: class diagrams, object diagrams, component diagrams and deployment diagrams. In addition, they could be used at statechart diagrams.

The union operation is the adding of two diagrams together. This seems quite simple but some problems can arise when the same element is present in both of the diagrams. The intersection is the corresponding part of two diagrams: the elements which are present in both of them. Then the last operation, the difference, is what is left when leaving out the intersection part of the diagrams.

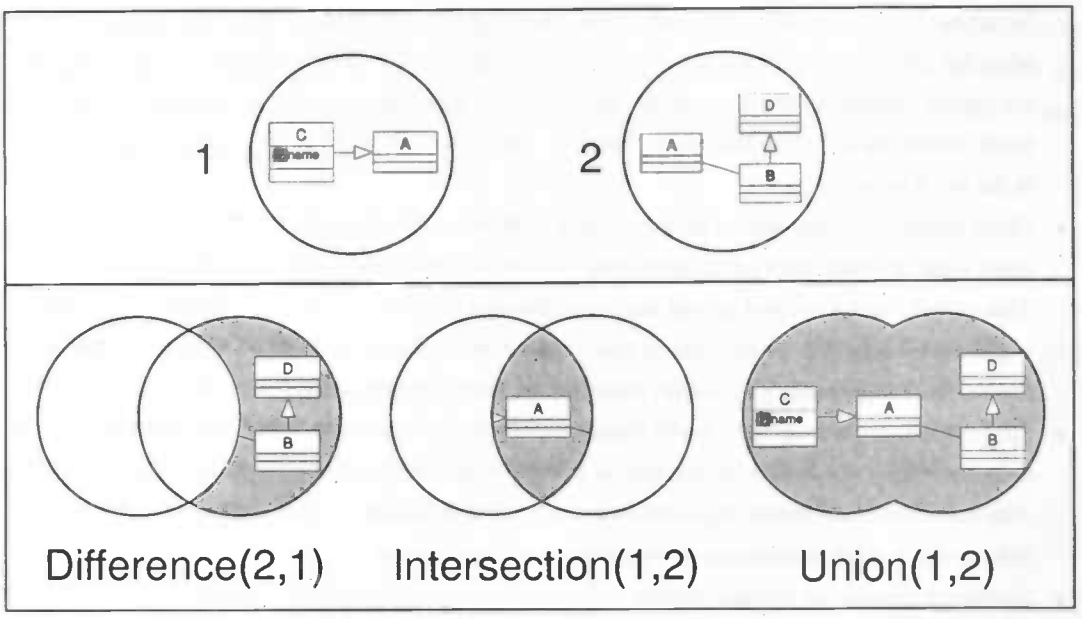

Figure 3.1: The three different set operations on class diagram fragments

In Figure 3.1 the three set operations are visualized on class diagrams. The diagrams are put into Venn diagrams to show the resemblance to set theory. The very simple models 1 and 2 (top of picture) are used in all the operations. The darker parts in the diagrams shows the range of the set operations. Note that the difference and union operations shown are symmetric operations. The difference operation is more useful for

diagrams when containing only one of the difference parts, so being asymmetric.

This thesis discusses the union, intersection and difference operations. An implementation of the set operations is explained as well as tested. The next section describes the thesis scenarios. These place the set operations in a software engineering context which can be validated.

3.3 Scenarios

For the desired manipulations mentioned above: merging, checking, and slicing of models, three scenarios are stated. The synthesis of models is left out here, because this is a manipulation which cannot be solved with the set operations but with the transformation operation [Sel03b]. The scenarios are all part of one of the desired manipulations, but they are more concrete and they are potentially solvable with a correct implementation of the set operations. Every description of a scenario is followed by a short description of test cases, which are concrete solvable situations of the described scenario. They will be validated in the validation section (Chapter 6). The concrete situation and the description of the used models will be given there too.

Scenario 1: Merging of models.

Different design teams have developed different parts of the system. These parts must be merged together to create a model containing the functionality of both models. Are there any inconsistencies? The set operations can perform the merging with the *union* operation and the *difference* operation can give clues to potential inconsistencies. In the test cases two forward engineered models from similar background, with much communality, will be merged together with the implemented union operation.

Scenario 2: Comparing different versions of a system.

A software system has been developed in an incremental way. UML models from different versions of the system are available, or different versions of the system are reverse-engineered into models. For an analysis of the design process, the different versions of models are compared. Three situations will arise: the checking of two reverse engineered models of different versions against each other, the checking of a forward engineered model against a reversed engineered model and the checking of two forward engineered models. The set operations can do this by visualizing the difference of two models, showing what has been changed in them. Two concrete test cases will be considered. One will check two reverse-engineered models against each other, the other will check a forward engineered model against its reversed counterpart. The conclusions can say something about the models as well as the model construction process.

Scenario 3: Finding and visualizing behavioral slices in class diagrams.

After a system is finished, models are not useless. Because of the potential reuse of the models, studying them is interesting. So, for a certain use-case data is enquired, for example from a behavioral diagram or from monitoring the actual use directly from the system. This data is transformed to class diagram parts. These parts can then be sliced from the model of the system by the set operations. These slices give a better insight in which parts of the system are affected by the behavior. The intersection operation can be used to show

the slicing results. In the test case, traces from a real functioning system are transferred to class diagrams. These class diagrams, containing classes and dependencies, are mapped on a reversed engineered model of the system. The traces are put into the context of the model and the impact can be observed.

3.4 Thesis questions

The scenarios and their test cases are used to evaluate the technique described in this thesis. All of these scenarios include some questions for the set operations. The main questions to be answered are:

- Can the set operations for UML class diagrams be implemented in an efficient, scalable, and usable way?
- What kind of the user influence is required for the set operations?
- Is it possible to create a general implementation which can be used without knowledge of the context of the models?
- How can the set operations be exploited for the different types of UML processing operations (merging, checking and slicing)?
- What is a useful way of reporting or visualizing the data enquired from the set operations?

The next Chapter will give the theoretical construction of the set operations followed by a description of the implementation in Chapter 5. In Chapter 6 test results will be discussed and the above described test cases will be executed and evaluated. In the last Chapter some concluding remarks are made as well as some recommendations for further research.

4 Problem Analysis

4.1 Basic Definitions

The definitions for model, diagram and diagram type are adopted from [Sel03b]. A *model* is a UML metamodel instance, a *diagram* is a graphical representation of the model, and a *diagram type* is a particular kind of diagram described by the UML Notation Guide.

UML metaclass instances will be addressed as *model elements* or as *elements* in this document. Every model element can have a certain *state*, which is defined by its *properties*. The properties of a model element are defined by the meta-attribute instance of that element. The *type* of an element is its metaclass. The names of the UML metaclasses are always written with capitals.

Connections between model elements are defined as follows: a *link* is a meta-association instance, connecting two model elements. A typical link for all model elements is a *parent* connection. The parent connection is defined through a special meta-association relationship, the composition relationship. Each model element can have at most one parent, as outlined in the MOF specification [OMG02].

A UML model is defined by all the model elements it contains, with the links these elements have with each other. A *model fragment* is a subset of a model, which can be handled as a model.

4.2 Correspondence

4.2.1 Equality vs. Correspondence

Theoretically set operations can be applied to all different types of elements. When at least one element is present in two sets, the sets have an *overlap*. Consider for example within the universe of all natural numbers between the one and twelve $\{1..12\}$ the two sets $Div2 = \{2, 4, 6, 8, 10, 12\}$ and $Div3 = \{3, 6, 9, 12\}$. Clearly only the elements 6 and 12 will be in both sets, so they are the overlap of those sets. This overlap is crucial for operations on sets. When determining the union of the above mentioned sets, the overlapping elements 'merge' together creating the resulting set: $(Div2 \bigcup Div3) = \{2, 3, 4, \mathbf{6}, 8, 9, 10, \mathbf{12}\}$. An intersection will contain all (and only) the overlapping elements, again merged together: $(Div2 \bigcap Div3) = \{\mathbf{6}, \mathbf{12}\}$. The difference will contain all but the elements of the overlapping part: $Div2 - Div3 = \{2, 4, 8, 10\}$ and

$Div3 - Div2 = \{3, 9\}$. So with the knowledge of the overlapping elements, all of the three basic operations can be calculated. Note that set operations on mathematical elements are easy because mathematics implies strict definitions for equality of elements.

Coming back to set operations on UML models, it is usually impossible to state that two model elements from different (parts of) models are equal. From the practical point of view this equality should be weakened. To do this, the concept of a *correspondence relationship* is introduced. The correspondence relationship is a zero-to-many relationship between model elements. Two model elements are considered to be *uniquely correspondent* when both elements have only one correspondence relationship, to each other. When two model elements are correspondent, they are assumed to be in the overlap part of the sets, representing the same semantic concept.

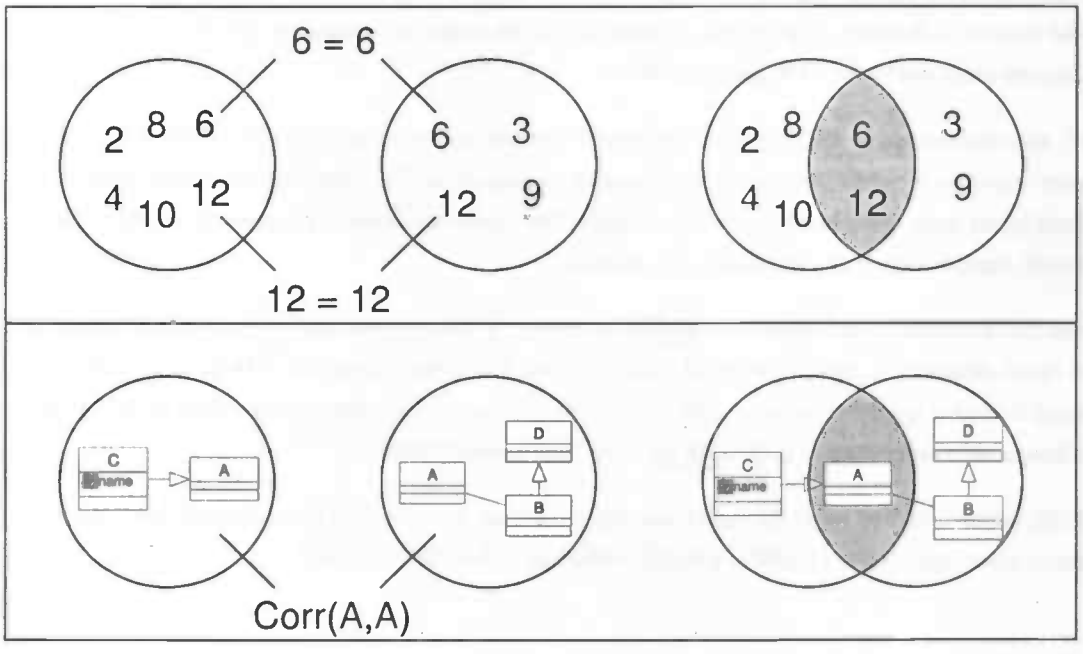

Figure 4.1: Equality and correspondence

A correspondence relationship is defined by *correspondence criteria*. These criteria can be applied on elements to determine if they have a correspondence relationship. They can vary with different types of models. They shall be discussed and defined for a part of the UML metamodel describing the basic class diagrams in section 4.3.

4.2.2 Union, Intersection and Difference

Assuming the existence of a correspondence relationship, the set operations can be defined. The following definitions assume two input models A and B. Also a merging method is assumed, which generates one element from two corresponding elements.

Definition 4.1: The *intersection* of models A and B consist of only the corresponding model elements, merged together.

Definition 4.2: The *union* of models A and B consists of all the elements from A and from B, where the corresponding elements are merged together.

Definition 4.3: The *difference* of models A and B is defined as all the non-corresponding elements from A.

Note that this definition of difference is an asymmetrical difference. All the elements from model B are lost, only the difference from model A is considered.

4.2.3 General criteria for UML diagrams

Specific correspondence criteria are needed to estimate the correspondence relationships. There are some general criteria which can be applied to MOF based systems, as described in [Sel04]. These criteria are defined at meta-metamodel level. By selecting a specific MOF metamodel, in this case the metamodel of UML, specific criteria for correspondence can be created based on those general criteria. These specific criteria can then be applied on a concrete UML model. The following general criteria should be considered for MOF based metamodels:

- *State criteria*: the first aspect of the state is the type of an element. In addition, an identifier for that element can be used, like the name or a repository id.
- *Parent criteria*: every element has a parent. For elements to have a correspondence relationship, it is required that they have correspondent parents.
- *Mandatory neighbour criteria*: these are the necessary connections of elements. The mandatory neighbours of elements must have a correspondence relationship for elements to have a correspondence relationship. This criterion is mostly used for the Relationships, who always need something to relate to.

The difference between the parent and mandatory neighbour is that all elements have a parent, but only some have mandatory neighbours. These general correspondence criteria are made specific by applying them on a certain part of the UML metamodel. The model is defined at metamodel level, and the rules can be created from that metamodel level. More criteria can be added, but by applying these basic criteria the general structure of the model is taken into account.

4.3 Class diagrams

4.3.1 The used model elements and the parent relationship

This section closely examines the above mentioned correspondence criteria for a selected subset of the UML metamodel. This subset contains elements which are sufficient to handle basic class diagrams. The following elements from the UML metamodel are used: Class, Interface, Attribute, Operation, Dependency, Association, AssociationEnd, Generalization. Figure 4.2 shows a subset of the UML metamodel concerning these elements:

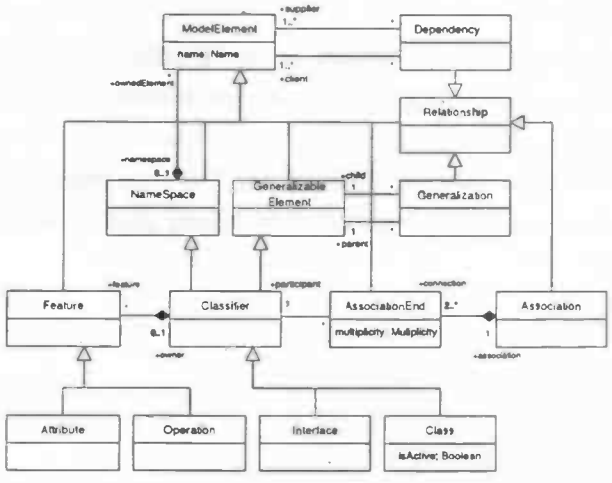

Figure 4.2: A subset of the UML metamodel

The Object Constraint Language (OCL) [OMG03, p.6-1] will be used to define the correspondence criteria. Because OCL lacks an attribute which holds the parent, it is defined here. This attribute is defined at metamodel level. All metaclasses have a parent, e.g. another metaclass which 'owns' the element. The parent relationship is visualized in the metamodel as a composition relationship. In figure 4.2 all of the compositions are shown for the elements which are discussed in this section. The well-formedness rules define that every element has at most one parent. So an AssociationEnd will have an Association as its parent, overwriting the Namespace. The parent links are differently named, but the implication of the parent relation is the same. The next definition defines the PARENT attribute:

With the attribute for the parent added, the definition of correspondence can be given. The correspondence criterion is built of different parts. It is built of a state correspondence criterion (`stateCorr`), a parent correspondence criterion (`parentCorr`) and a mandatory neighbour correspondence criterion (`mNeighbourCorr`). Definitions of these criteria are given in the next sections.

```
-- Definition of PARENT attribute
Context ModelElement
def : PARENT : ModelElement = self.namespace
Context Feature
def : PARENT : ModelElement = self.owner
Context AssociationEnd
def : PARENT : ModelElement = self.association
```

Figure 4.3: The definition of PARENT

4.3.2 State criteria

The state criterion discussed here consists of a type and a name criterion. The type of a model element is the first part of the state correspondence criterion. Every model element has a metaclass. For two model elements to be corresponding, they must come from the same metaclass. This results in the following OCL definition for type correspondence:

```
-- Type correspondence criterion
let typeCorr (e1: ModelElement, e2: ModelElement): Boolean
    = e1.oclIsTypeOf(e2.evaluationType())
```

Figure 4.4: The type correspondence criterion

In this formula `typeCorr` represents the *type correspondence criterion*. Note that the metaclass of a model element can have other metaclasses as its ancestors. Consider the metamodel fragment in Figure 4.2, it shows that for example Generalization has ancestors Relationship and ModelElement. This can be used to construct variations for type correspondence. For example, elements with common ancestors can be considered correspondent. So when two model elements are both Relationships they can be considered corresponding, although one is a Dependency and the other is a Generalization.

```
-- Type-relationship correspondence criterion
-- Example of extended type correspondence criterion
let typeRelCorr (e1: ModelElement, e2: ModelElement): Boolean
    = (e1.oclIsTypeOf(e2.type) or
        (e1.oclIsKindOf(Relationship) and
         e2.oclIsKindOf(Relationship) ) )
```

Figure 4.5: The type-relationship correspondence criterion

As defined in the UML specification every ModelElement has a property *name* (see the metamodel fragment in Figure 4.2). Assuming that names are constructed in a meaningful way, they are used as a second state criterion for correspondence, as is shown in Figure 4.6.

In this formula, `nameCorr` represents the *name correspondence criterion*. This criterion is very strict be-

```
-- Name correspondence criterion
let nameCorr (e1: ModelElement, e2: ModelElement): Boolean
    = (e1.name = e2.name)
```

Figure 4.6: The name correspondence criterion

cause the complete equivalence is required for the names. At implementation level this could be weakened to case-insensitive comparing of names, or to handle different naming conventions. This criterion can also be strengthened when the model elements have a unique id which can be used as a criterion. Combining the definitions for the type and the name criteria results in the *state correspondence criterion* (`stateCorr`):

```
-- State correspondence criterion
let stateCorr (e1: ModelElement, e2: ModelElement): Boolean
    = (typeCorr(e1, e2) and nameCorr(e1, e2))
```

Figure 4.7: The state correspondence criterion

This is a very simple definition of the state criterion. Potentially all properties could be checked under different circumstances to influence the state correspondence outcome.

4.3.3 Parent criterion

Two elements can only have a correspondence relationship when their parents also have a correspondence relationship. The parent function is defined above, so every used element has a `PARENT`. This is used to construct the *parent correspondence criterion* (`parentCorr`).

```
-- Parent correspondence criterion
let parentCorr (e1: ModelElement, e2: ModelElement): Boolean
   = if (e1.PARENT.oclIsUndefined() or
         e2.PARENT.oclIsUndefined() )
       then
          e1.PARENT.oclIsUndefined() and
          e2.PARENT.oclIsUndefined()
 else
          corr(e1.PARENT, e2.PARENT)
 endif
```

Figure 4.8: The parent correspondence criterion

This definition implies recursion, because corr is used, while in the definition of corr the parent correspondence criterion is used. This criterion is not symmetric in the way that the correspondence of the children does not influence the correspondence of the parents. So for example Classes can be correspondent only when their parent is correspondent, but its Features don't have to be correspondent too.

Because model elements can have an undefined parent, the first part of the definition is needed. When two parents are both undefined, they are assumed to be corresponding. When one of them is undefined and the other is not, they are considered non-corresponding.

4.3.4 Mandatory neighbours

Some elements require the existence of other elements, in addition to the parent. Mandatory neighbours can be found in the metamodel, when there is a meta-association with the multiplicity lower bound greater than zero, as defined in [Sel04]. This occurs mostly with Relationships, because they require something to relate to. For two model elements to be correspondent it is needed that their mandatory neighbours are also correspondent.

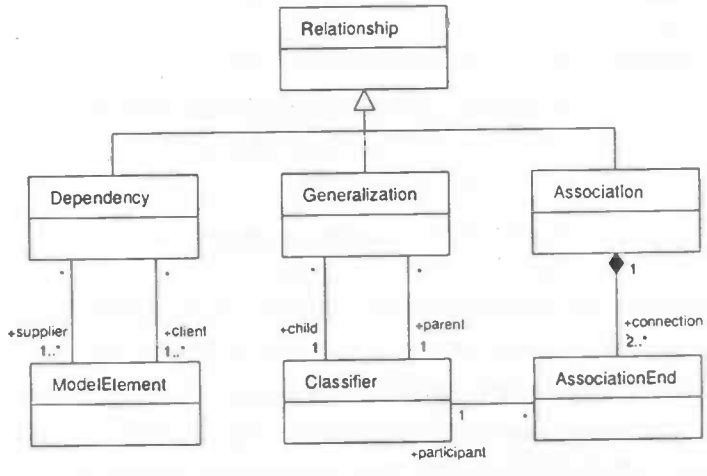

Figure 4.9: A metamodel fragment of the Relationships

In Figure 4.9 a part of the metamodel structure of UML, focusing on Relationships is shown. Generalizations have two links to Classifier, the child and the parent. Both of the links have a multiplicity of one. This means that the Generalization cannot exist without at least two connections to Classifiers. This results in the *Generalization correspondence criterion* (genCorr):

```
-- Generalization correspondence criterion
let genCorr (e1: Generalization, e2: Generalization): Boolean
    = (corr(e1.child, e2.child) and
       corr(e1.parent, e2.parent) )
```

Figure 4.10: The Generalization correspondence criterion

Now the Dependency causes a slight problem, because it can contain *at least* one client and supplier, but there can be more. For this case it is necessary to iterate over every client and supplier to check if they have

corresponding elements. This results in the *Dependency correspondence criterion* (Figure 4.11).

```
-- Complete Dependency correspondence criterion
let complDepCorr (e1: Dependency, e2: Dependency): Boolean
    = (e1.supplier.forAll(s1|
         e2.supplier.exists(s2|corr(s1, s2) ) ) and
       e2.supplier.forAll(s2|
         e1.supplier.exists(s1|corr(s2, s1) ) ) and
       e1.client.forAll(s1|
         e2.client.exists(s2|corr(s1, s2) ) ) and
       e2.client.forAll(s2|
         e1.client.exists(s1|corr(s2, s1) ) ) )
```

Figure 4.11: The complete Dependency correspondence criterion

```
-- Simplified dependency correspondence criterion
let depCorr (e1: Dependency, e2: Dependency): Boolean
    = (corr(e1.supplier[0], e2.supplier[0]) and
       corr(e1.client[0], e2.client[0]))
```

Figure 4.12: The simplified Dependency correspondence criterion

When assuming only one client and one supplier for every Dependency the statement becomes more readable (Figure 4.12). This assumption is quite realistic, because in most of the cases the Dependency relates only two model elements. Analogous to the definition of the dependency correspondence criterion are the correspondence criteria for AsociationEnd and Association (Figure 4.13). The mandatory neighbour relation of Associations and AssociationEnds is bi-directional. This means that the correspondence criteria rely on each other. This should be taken into account when coming to implementation level.

```
-- AssociationEnd correspondence criterion
let assEndCorr (e1:AssociationEnd, e2: AssociationEnd): Boolean
    = (e1.participant = e2.participant)

-- Association correspondence criterion
let assCorr (e1: Association, e2: Association): Boolean
    = (e1.connection.forAll(c1|
         e2.connection.exists(c2|corr(c1, c2) ) ) and
       e2.connection.forAll(c2|
         e1.connection.exists(c1|corr(c2, c1) ) ) )
```

Figure 4.13: The AssociationEnd and Association correspondence criteria

With the above defined definitions, the *mandatory neighbour criterion* of this subset of the UML model can

be defined (see Figure 4.14).

```
Context ModelElement
Def : mNeighbour (e2: ModelElement)    : Boolean
      = TRUE

Context Generalization
Def : mNeighbour (e2: Generalization) : Boolean
      = genCorr(self, e2)

Context Dependency
Def : mNeighbour (e2: Dependency)      : Boolean
      = depCorr(self, e2)

Context AssociationEnd
Def : mNeighbour (e2: AssociationEnd) : Boolean
      = AssEndCorr(self, e2)

Context Association
Def : mNeighbour (e2: Association)     : Boolean
      = assCorr(self, e2)

-- Mandatory neighbour correspondence criterion
let mNeighbourCorr (e1: ModelElement, e2: ModelElement): Boolean
    = e1.mNeighbour(e2)
```

Figure 4.14: The complete mandatory neighbour correspondence criterion

4.3.5 Correspondence

The three main criteria defined above (`stateCorr`, `parentCorr` and `mNeighbourCorr`) result in the complete criterion for correspondence (`corr`).

```
-- The correspondence criterion
let corr (e1: ModelElement, e2: ModelElement): Boolean
    = (stateCorr(e1, e2) and parentCorr(e1, e2) and
       mNeigbourCorr(e1, e2) )
```

Figure 4.15: The correspondence criterion

By applying this criterion to concrete model fragments correspondence relationships can be calculated. The criteria defined above are not very strict and can be changed when handling different diagrams. Take for

example the name rule; this can be changed to a name rule which checks for case-insensitive correspondent names. Or consider, for a specific system, there is the need to also check if the Classes of the model have the same Attributes. This could be used as extra rules, and are to be defined by the users. It could even be necessary to lose some of the criteria. When one of the models does not have a package hierarchy it could be necessary to relax or even dismiss the parent criterion.

4.3.6 Using the correspondence criteria

To show what the correspondence criterion does on real model fragments, consider the example fragments from Figure 4.16. They are very simple models, but they are sufficient to show the working of correspondence. In figure 4.17 the metamodel instances of the two models are shown.

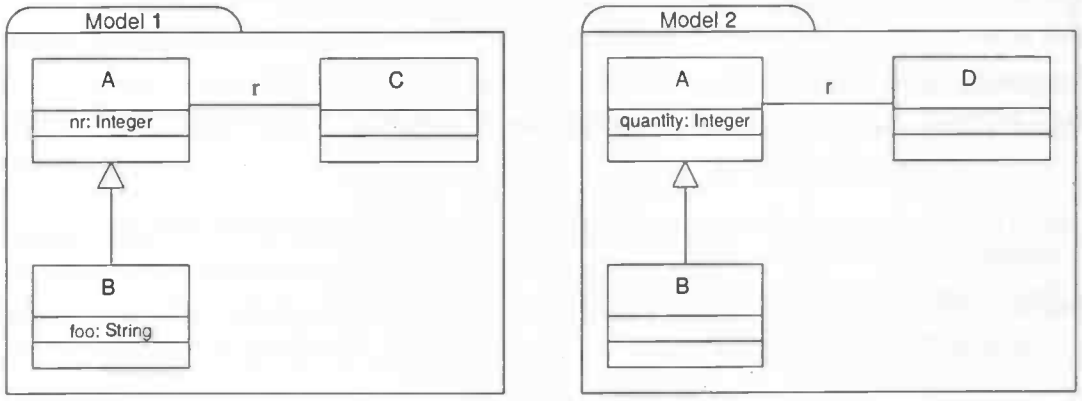

Figure 4.16: Two example models

In figure 4.17 the corresponding metamodel instances are darker. Class A, B, and the Generalization between them are corresponding elements. The two attributes from Class A, nr and quality, have different names, and are therefore not corresponding to each other. The Association seems quite similar, but the lower AssociationEnds are not corresponding because they have non-corresponding mandatory neighbours: Class C and Class D. This is why the Associations are also non-corresponding (they depend on their mandatory neighbours, the AssociationEnds). Because this Association is non-corresponding the top AssociationEnds cannot be corresponding. The Attribute of Class B has no potential corresponding element from the other model, so it is none-corresponding.

4.4 From correspondence to Set Operations

A problem arises when going from equality to correspondence as criterion to determine the overlap of two sets. When dealing with equal elements, it does not matter which element is used from the overlapping part. But when two elements are correspondent, which of them should be used as a representation of them? A

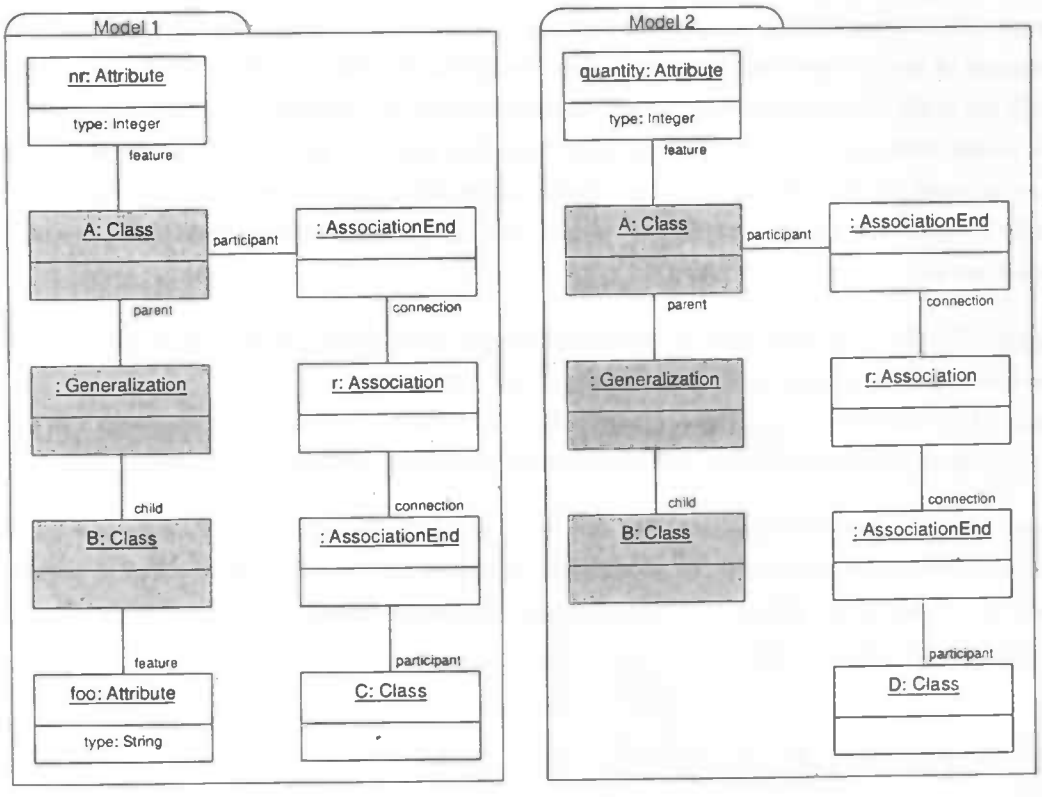

Figure 4.17: The metamodel instances of the two example models

method is needed to merge these two elements together. Not all the properties of an element are used for the correspondence calculation. Two corresponding elements usually have different states, for example they can have different comments. The most straightforward solution is to favor one of the elements over the other, and use it for the result. This means that the set operations will be asymmetric. The user should be well aware of this and needs to be able to influence which element is to be chosen as a representation.

With the knowledge of the corresponding elements and a way of merging elements the set operations can be constructed. The union, as defined in definition 4.1 is somewhat complicated. The links need to be reconnected, when it is a connection between a corresponding and a non-corresponding element from model B. In the example, the Classes A and B and the Generalization between them is taken from model 1, the AssociationEnd and the Attribute quantity are reconnected to Class A from model 1. Second, the intersection; this consists of the corresponding elements, merged together (as described in definition 4.2). In the used example this results in the Classes A and B, and the Generalization between them. The difference operation shows another challenge, since it can produce model fragments which are not well formed. As can be seen from the example, the two Attributes as well as the top AssociationEnd are not connected to anything anymore when the corresponding elements are deleted. To solve this two options are offered (again using two input models A and B):

Definition 4.3a: The *Destructive difference* contains all the non-corresponding elements from model A. Any connections to model elements which do no longer exist are discarded.
Definition 4.3b: The *Preserving difference* contains all the non-corresponding elements from model A as well as the corresponding elements from A which are necessary to maintain the well-formedness of the model.

In figure 4.18 the Intersection, Union and two differences are shown on the example from figure 4.16. In this figure the results are visualized as separate models. In different situations it might be useful to visualize the results as a part of one or both of the input models. Or it might be required to report the results in a none-graphical but statistical way.

4.5 Related work

The essentials for the theory used in this thesis were developed by Selonen [Sel03b]. Selonen addresses the set operations on a smaller subset than used in this thesis. The concept of correspondence and preserving / destructive difference is taken from there. The correspondence criteria are not as concretely defined as in this thesis, and no implementation is given. The concepts of the state, parent and mandatory neighbour correspondence are defined at MOF level in [Sel04]. They are the foundation for the UML state, parent, and mandatory neighbour criteria described above.

Much literature discusses the usefulness of using merging and slicing operations. MDA is highly dependent on the merging of models [OMG03c, p.3-11]. The merging of the contents of packages is discussed in the UML version 2.0 [OMG04, p.155].

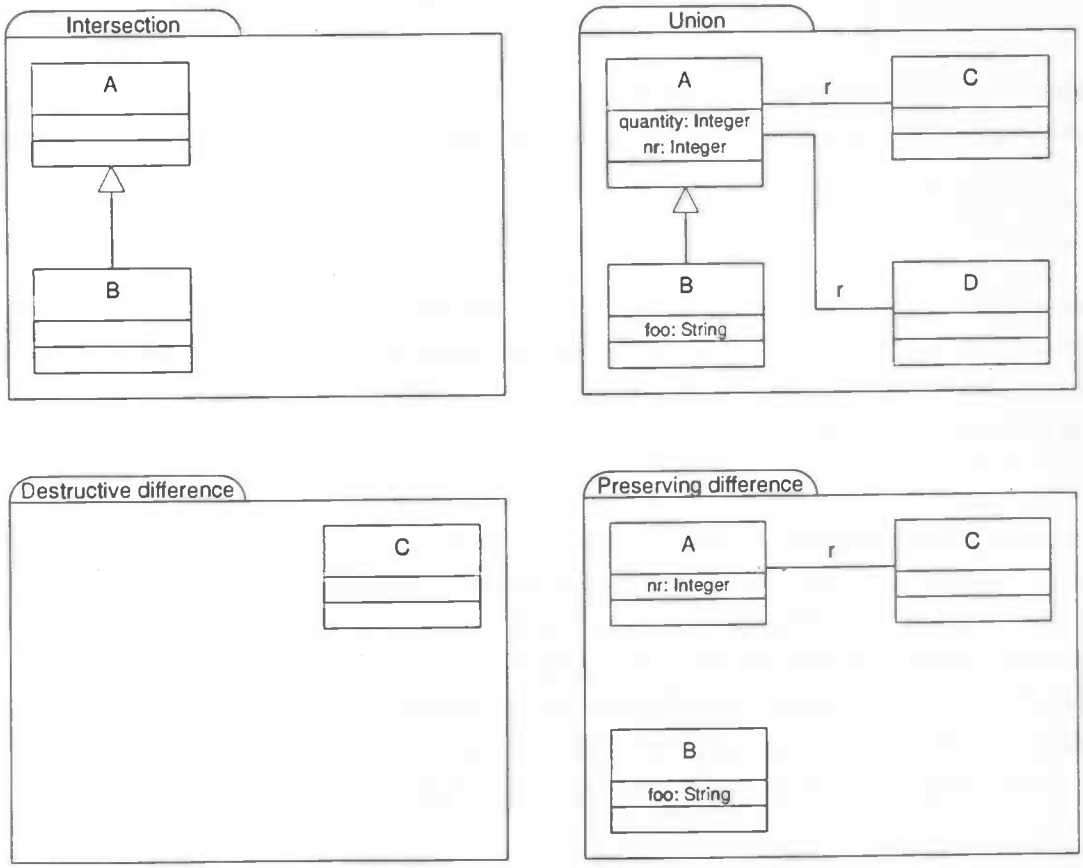

Figure 4.18: The result of the example models

Concrete implementations of Set Operations are not very common. In [Ohs03] differences between diagrams are discussed, in the context of a versioning control system for models. Special focus here was on the visualization of the differences and similarities. Repository identifiers are used for detecting equal elements, in contrast to the here described correspondence detection criteria. For reengineering purposes the set operations are often referred to. In [Kol02] reverse-engineering techniques are compared with a simple implementation of set operations, using only the name for comparison. [Ala03] presents a method for calculating union and differences of models, but it is again highly dependent on the existence of a unique identifier for the calculation of equal elements.

5 Implementation

5.1 Working Environment

This section discussed the platform where the set operations are implemented on. First the background and general information is given followed by a short explanation of the data representation on the used platform.

5.1.1 xUMLi

The set operations are implemented on the xUMLi platform [xUMLi]. This platform is developed at the Institute of Software Systems of the Technical University of Tampere (TUT), in the ART project [Art03]. This platform is used to manipulate UML models. The basic idea is to create a set of simple manipulation operations which can be used after each other to create more complex functionality. The set operations are one of the desired operations for this platform.

Models from different CASE tools and model formats can be used to import and export data to and from the xUMLi platform. The xUMLi *operations* are used to manipulate the models. The operations, as well as the importers and exporters, are implemented as COM-components, typically as Python [Pyth] or C++ components. The xUMLi platform is constructed on the scripting engine VISIOME. VISIOME was also developed at the TUT, specified by Jari Peltonen [Pel00] and implemented by Mika Siikarla [Sii02]. VISIOME is a data-independent scripting engine, which in the case of xUMLi processes UML model data. For a simplified view of how this platform works in interaction with other CASE tools or UML file formats see figure 5.1.

An important aspect of the xUMLi platform is the CASE-tool independence. Import and export components for specific CASE tools are used to convert the specific UML data to and from xUMLi data. For this project mainly the tool Rational Rose [Rat04] is used because it has been applied with xUMLi before and the import and export modules for Rational Rose have been constructed and tested. The operations will however be usable for models from potentially all CASE tools, when a specific import and export component in xUMLi is constructed.

Within the xUMLi environment Python or C++ components can be added. All the components have at least one input and one output data stream. Within the components the data streams can be manipulated and new streams can be constructed. Also, the streams can be explored and used for the analysis of models. The data streams can be exported as a UML models of different formats with the xUMLi exporters.

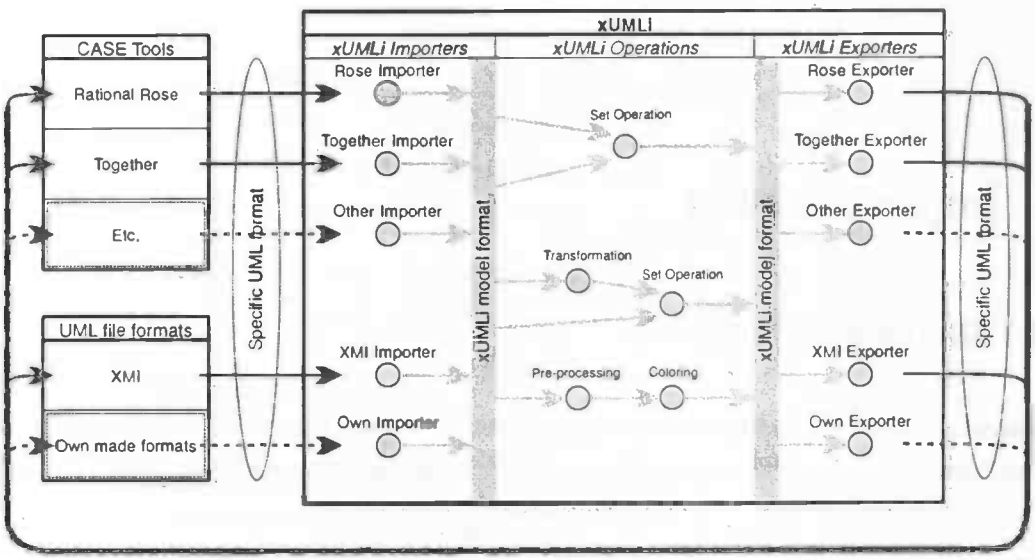

Figure 5.1: Data flow for xUMLi and external CASE tools and file formats

The construction and connection of the components is done with XML-formatted run scripts. Two types of connections (streams) are used. The *control stream* handles the order in which execution take place, and the *data stream* defines the data-flow. The VISIOME Editor can be used to construct run scripts in a visual way. Figure 5.2 shows an example of a visual construction of a script. The actual run script which is generated from the Editor is quite big and unclear to read, it is not shown here.

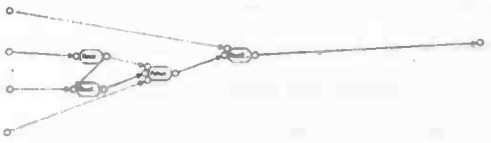

Figure 5.2: An example of a visual constructed script in the VISIOME Editor

The set operations are constructed as a component which is used in such run scripts. This component has three input data-streams and one output data stream (similar to the component in the middle of figure 5.2). The first two streams contain two models on which the operation functions, and the third one contains control data for determining the rules of the set operation. The output stream contains the result of the set operation.

5.1.2 The data structure of xUMLi

The data representation in the xUMLi platform directly reflects the metamodel structure of UML. The input streams contain vectors, which contain ModelElements. The elements are connected with each other as

described in the UML metamodel. In addition to the standard *UML defined connections* some functions are implemented for easy access to other elements. There are for example functions for getting the parent (owner, namespace or association; dependent on the element) and the type (metaclass) of an element. Also every element has an unique identifier, the HashValue. This identifier uniquely identifies elements within the xUMLi system, so they cannot be used outside this system.

As an example, figure 5.3 shows a simplified part of the xUMLi data structure. In figure 5.4 the same elements are shown, taken from the UML specification. This clearly shows that the data structure of xUMLi follows the data structure of the UML metamodel directly.

```
abstract element ModelElement inherits VisioMeElement
{ (0..1)  name String
  (0..*)  supplierDependency --> [supplier] Dependency
  (0..*)  clientDependency --> [client] Dependency
  (0..*)  comment --> [annotatedElement] Comment
  (0..1)  namespace within [ownedElement] NameSpace
  (0..*)  viewElement --> [base] UmliViewElement }

relation Dependency inherits Relationship
{ (1..*)  supplier --> [supplierDependency] ModelElement
  (1..*)  client --> [clientDependency] ModelElement }

abstract element Classifier
          inherits NameSpace; GeneralizableElement
{ (0..*)  feature /*[owner]*/ Feature
  (0..*)  association --> [participant] AssociationEnd
  (0..*)  instance --> [classifier] Instance }

relation AssociationEnd inherits ModelElement
{ (0..1)  isNavigable Boolean
  (0..1)  multiplicity Multiplicity
  (0..1)  participant --> [association] Classifier
  (1..1)  association within [connection] Association }

relation Association inherits Relationship;GeneralizableElement
{ (2..*)  connection /*[association]*/ AssociationEnd
  (0..*)  link --> [association] Link }
```

Figure 5.3: A part of the xUMLi data structure

For example the ModelElement from figure 5.4 has a name and connections to Dependency, the clientDepen-

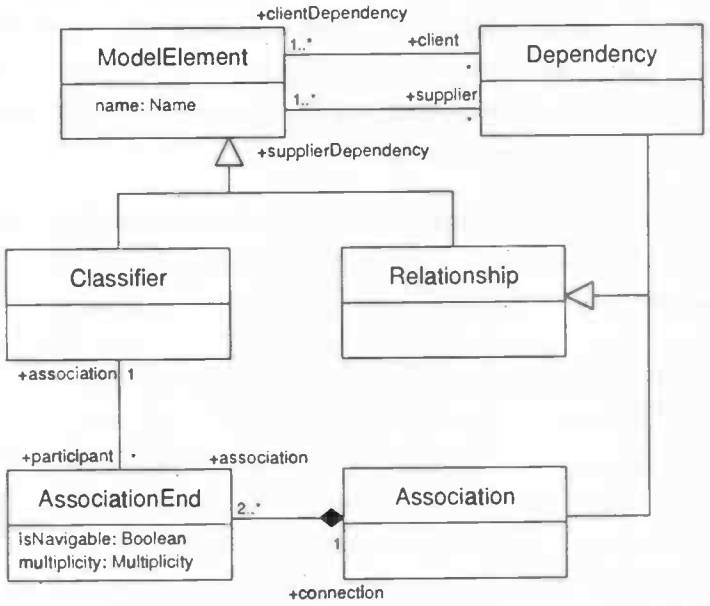

Figure 5.4: A UML metamodel fragment (simplified from [OMG03, p. 2-14/15])

dency and the supplierDependency. These connections as well as the string for the name are accessible in the xUMLi system. In figure 5.5 an example is shown of how this data structure can be used. In this example the

```
# Example of the xUMLi data system:
# The variable element is assumed to be a ModelElement

clientDependency = element.Get("clientDependency").GetAt(0)
otherClassifier = clientDependency.Get("supplier").GetAt(0)
print "Name of other Classifier is "
              + otherClassifier.Get("name")
```

Figure 5.5: An example of the xUMLi data system

Dependency which is connected to `element` is taken (first line). The other connection (the supplier) of that Dependency is found (second line), and the name of that element is printed (third line). The GetAt statement is needed because the connections have a vector as a result. `GetAt(0)` gives the first element of that vector.

5.1.3 Exploring and manipulating the xUMLi data

Within the xUMLi platform OCL statements can be executed on xUMLi elements. The statements can be combined with the programming language which is used [Sii02]. Figure 5.6 shows an example of an OCL

expression in Python on xUMLi elements.

```
# Example of the xUMLi Select statement in Python:

dependencies = package.Select("element.metaclass->exists
                                (mc|mc='Dependency')")
```

Figure 5.6: An example of an OCL statement in xUMLi

As can be seen, the resulting `dependencies` contains all the elements owned by the `package` which have metaclass Dependency. In addition to the standard OCL *Select* expression the *Find* expression is implemented on the xUMLi platform [Sii02]. This expression is similar to the Select statement but it searches the whole ownership hierarchy recursively.

Information can be acquired from xUMLi elements with the Get statement, and elements can be changed with the Set statement. For example in the next statement (Figure 5.7) all the elements with metaclass Classifier are selected. There is iterated over all of them and the name of the element and its parent are printed. When the element with name `MainClass` is found, a line to the comment of that element is added.

```
# Example of the xUMLi Find and Get statements in Python:

classifiers = package.Find("element.metaclass->exists
                              (mc|mc='Classifier')").ToList()
for c in classifiers:
  print c.Get("name") + " parent: " + c.Parent.Get("name")
  if (c.Get("name") == "MainClass"):
    c.Set("comment", c.Get("comment") + " main class")
```

Figure 5.7: An example of the xUMLi Find and Get statements

With the OCL statements and the Get and Set functions the xUMLi model can be explored and manipulated. This is used in the implementation of the set operations, as described in the next section.

5.2 Implementation of the Set Operations

5.2.1 Overview

The set operations are constructed in two phases. First the correspondence is calculated. This means that the elements of the two input models are compared to each other and the corresponding elements are marked. The correspondence calculation uses the correspondence rules to calculate the correspondence. The user can influence what is to be considered as corresponding by manipulating the rules and selecting several options. The correspondence calculation is described in section 5.2.2. The second phase of the set operations is the

result management. Here, the correspondence information is used to construct the set operations. The result management can also be influenced by the user, as described in section 5.2.3.

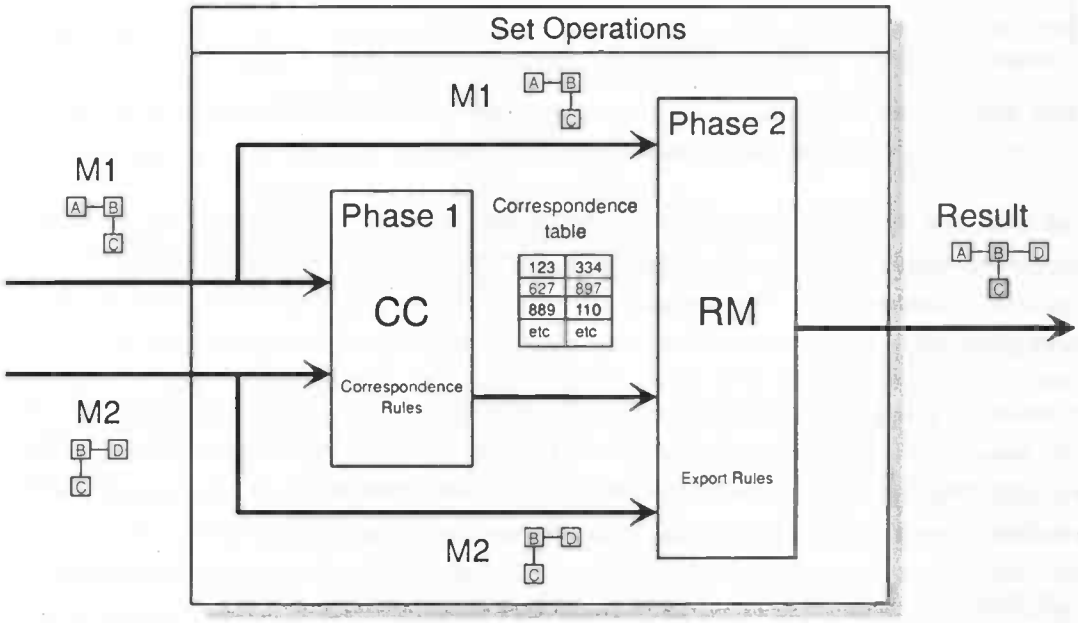

Figure 5.8: Creation of set operations

5.2.2 Phase one: Correspondence Calculation (CC)

The correspondence criteria from Chapter 4 are implemented as correspondence rules for the concrete comparing of elements. A *correspondence rule* is a function $M \times M \rightarrow B$, where M represents a model element, and B a Boolean result-value. Generally, two types of rules can be distinguished:

- *Internal rules.* These rules use only the information from the state of the element. Typically the properties of the element determine the outcome of the rule. Examples are: the comparing of name, type, multiplicity, etc.
- *External rules.* Rules for which the outcome is dependent of other elements linked to the inspected elements. Examples are: checking of the parent, its stereotype or necessary connections.

Referring to the criteria mentioned in Chapter 4, the internal rules are defined by the state criterion and external rules by the parent and mandatory neighbour criteria. In figure 5.9 implementations of rules are given, constructed from the criteria. The first rule compares the names of the elements (name criterion), the second one compares the parents of the elements (parent riterion), and the third one the necessary connections for the generalization (generalization criterion). The first one is an internal rule, the other two are external ones. Note that the external rules use the function `function` to compare the connecting elements. All the other rules are implemented in a similar way.

```
# Example of implemented rules.

# Name rule:
def NameRule(a, b):
    return (a.Get("name") == b.Get("name"))

# Parent rule:
def ParentRule(a, b, function):
    return function(a.Parent, b.Parent)

# Generalization rule:
def GeneralizationRule(a, b, function):
    return function(a.Get("child"), b.Get("child")) and
           function(a.Get("parent"), b.Get("parent"))
```

Figure 5.9: An implementation of some rules for correspondence detection

The results of internal rules are independent of other elements, so they result directly as a Boolean. Before using external rules it is necessary to mark the current elements as in-use, because it is possible that elements are circular dependent for their correspondence. (For example the AssociationEnd is dependent of the Association, and the Association is dependent on the AssociationEnd) This is solved by marking the elements as in-use before checking the external rules. When two elements are compared which are already marked against each other, they return true.

The correspondence rules have been implemented in a separate file as very simple Python functions. The rules are called form the main program by calling the functions ApplyInternalRules and ApplyExternalRules. These functions can be changed independent of the main program. New rules can be freely added in this file, or existing rules can be modified. For example a rule can be added for the checking of stereotypes.

In theory, every model element can be corresponding to every other element. The most straightforward method of comparing elements is just iterating over every element of the first model, and comparing them to every element of the second model. This is a useful implementation when there is completely no knowledge about the used models or the used correspondence criteria. However this is not very efficient and often a more efficient approach can be used.

When two model elements have found to be corresponding, they are marked against each other with a *corresponding mark*. The corresponding marks are saved in a hash table, the Correspondence table. This table is used as the result of the Correspondence Calculation and used in the Result Management.

In practice also the efficiency of the program must be taken into consideration. Especially when the models are large, it is very time-consuming to check all the model elements against each other. So a few enhancements have been applied to the comparing process:

- Only the interesting elements are checked (e.g. the elements of the subset which is discussed in chapter 4.3). The list of interesting elements can be set from the rules file, so it can be changed easily.
- One-to-one correspondence: when a corresponding element is found, this element is not checked against the rest of the elements anymore. So no more than one corresponding element can be found. This is no problem when assuming uniquely correspondent elements.
- The hierarchy is taken into account. First the elements of one level (namespace) are compared. Then the owned elements of the corresponding elements are checked. All the elements owned by non-corresponding elements are considered non-corresponding.
- The Relationships are only checked when corresponding Classifiers are found.
- Model elements which are marked as in-use are directly considered correspondent or not.

With these efficiency enhancements applied, the correspondence calculation worked within reasonable time. Disadvantage is that by applying these enhancements, the generality of the implementation is decreased, because some assumptions have been made about the input models and the calculation process.

For the correspondence calculation some options can be selected. All the used correspondence rules can be set on or off, by adjusting the flag for that rule (NameRule, TypeRule, ParentRule, DependencyRule, GeneralizationRule, AssociationRule, AssociationEndRule, StereotypeRule). Another flag can be set to let the set operation go through the hierarchy or not, GoThroughHierarchy. When this option is not set all the elements are brute-force checked against each other.

The result of the correspondence detection is a hash table which contains for every element an entry which points to its corresponding counterpart (if one exists). For the index and pointer values the HashValue is used. If an element is non-corresponding, the entry of that element points to -1.

5.2.3 Phase two: Result Management (RM)

For the result management the two input models and one hashtable which contains the correspondence data is needed. The hashtable is used to lookup the correspondence as well as to find the corresponding element when an element needs to be reconnected. According to which operation is desired (as described in Definition 4.1, 4.2 and 4.3a), the following action is taken in the Result Management:

- *Union*. The first input model is taken as the start of the result. All the elements from the second model are taken one by one, and checked if they are none-corresponding. All the none-corresponding elements are added to the result. When there are connections from none-corresponding to corresponding elements in the second model, the connections are reconnected to their counterparts of the first model.
- *Intersection*. For the intersection the first input model is taken as the start of the result. For every element of this model the correspondence is checked in the hashtable, if an element is none-corresponding, it is removed from the result. This leaves only the corresponding elements, which is exactly the result of the intersection.
- *Difference*. For the difference all the elements from the first model are checked in the hashtable and

the corresponding elements are removed. This way of working can produce un-well formed models. The exporter corrects this and cuts off the un-well formed connections. This means that the result will be the *destructive difference*.

In figure 5.10 the resulting operations are shown, very simplified. The cross represents deleted elements, only used at the intersection and the difference.

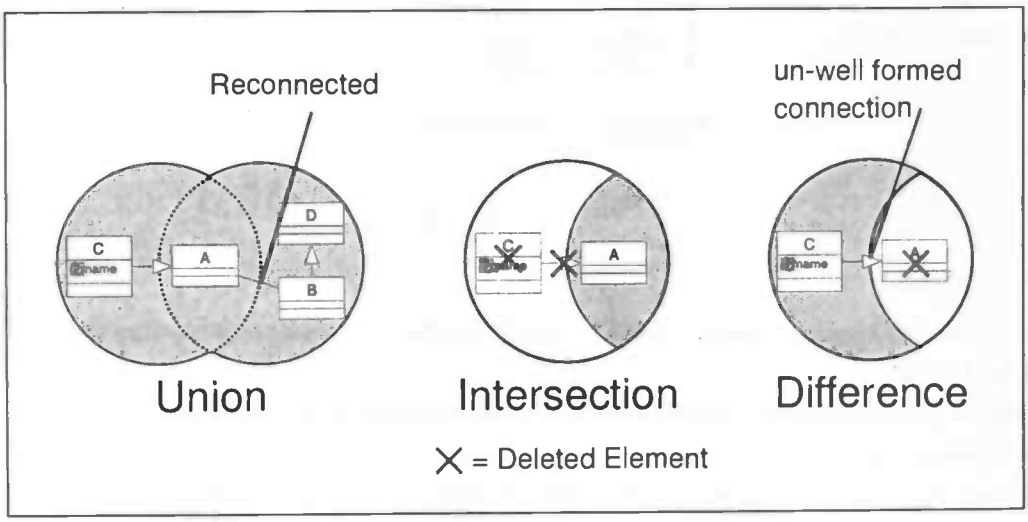

Figure 5.10: The result creation process

The using of colours has proved to be very valuable for visualizing the results. Especially when one of the models is completely outputted, and the result is only coloured. This is why the using of colours is integrated as a possibility to generate output. The Export Rules are determined by user defined options. The set operations can be called with the following options for the result management:

- Operation (String) Specifies what the resulting set operation is: Union, Intersection or Difference.
- AskUser (Boolean) If this flag is set, the user is asked at the end of the calculation for how the results need to be processed. The use of colors and the colors can be selected, as well as the use of a summary diagram. This GUI can be modified to include more options. Figure 5.11 shows an example of the implemented GUI.
- UseColours (Boolean) Check to use colours in the result. Different colours can be given for the three parts of the result (see colourA, colourB and colourC).
- MakeDiagram (Boolean) When this option is set, a summary diagram will be constructed with all the elements of the result.
- Name (String) The name of the resulting package.
- MakeDifferenceDiagram (Boolean) Creates a diagram, just like the MakeDiagram flag, but it only contains the elements which are none-corresponding and the direct corresponding links.
- ToDataFile (Boolean) A comma separated file with the data about the correspondence is created in addition to the resulting diagram.

Figure 5.11: A GUI example

- ColourA (three integers) Represent the RGB value of the colour for the none-corresponding elements in the first input.
- ColourB (three integers) Represent the RGB value of the colour for the none-corresponding elements in the second input.
- ColourC (three integers) Represent the RGB value of the colour for the corresponding elements.

5.3 About the implementation

The user interaction is static, it requires no used interaction except for the specified options and rules. The correspondence calculation rules are applied by the system without any knowledge of the models. Extra functionality could appear with the introduction of dynamic user interaction. In this way, the set operations will be able to ask the user to make a decision at a certain point in the correspondence calculation, or at time of the processing of the results. Knowledge which is not easy to specify in rules can then be used to influence the correspondence decision. The dynamic user interaction has been implemented, but it is not tested intensively because the test models did not require it.

At implementation stage the differences between diagrams and model began to emerge as a challenge. The differences were not very clear at every point; sometimes information about the complete model arose in a diagram. At other times diagrams became too big when they were created as results from operations on models. The current implementation can handle both diagrams and models as input, and can output them both too.

Using hash tables to mark the elements for correspondence. The constant manipulation and checking of the elements within the xUMLi system was causing the set operations to run slowly. By doing this with hash tables the execution time decreased with 80%. Disadvantage is that the hash table is not part of the xUMLi system so the results of the correspondence calculation are only useful within the set operations.

5.4 Example

In this section the example from the end of Chapter 4 is taken, and the results are shown. To recall, the models used are shown in figure 5.12, screen shots taken from Rational Rose. The results are best visualized with colors, but to give an idea, the union is shown in figure 2.13 in black and white. The Classes A and B have the same color, and also the Generalization between them, These elements represent the overlapping part. Class D with its Association has an other color, presenting that it comes from model 2. Class C is also colored differently, this comes from model 1.

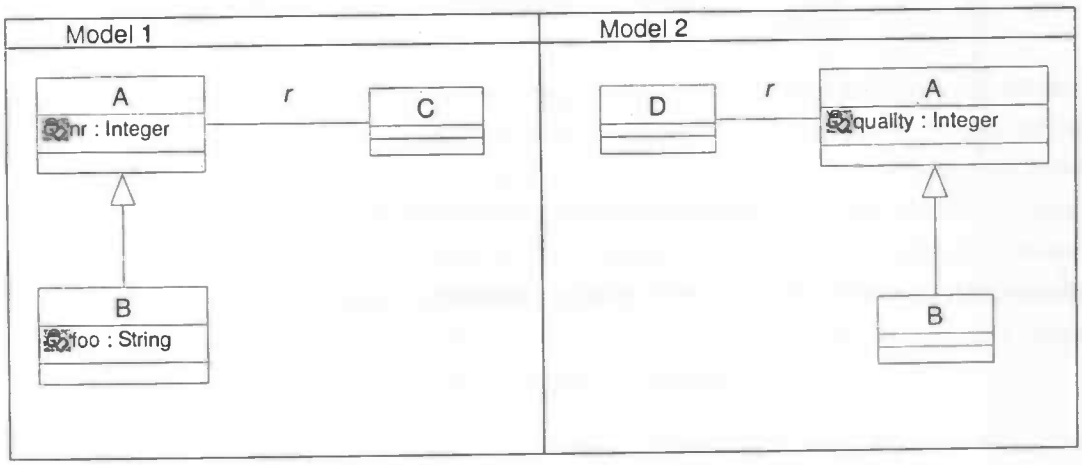

Figure 5.12: Two example models

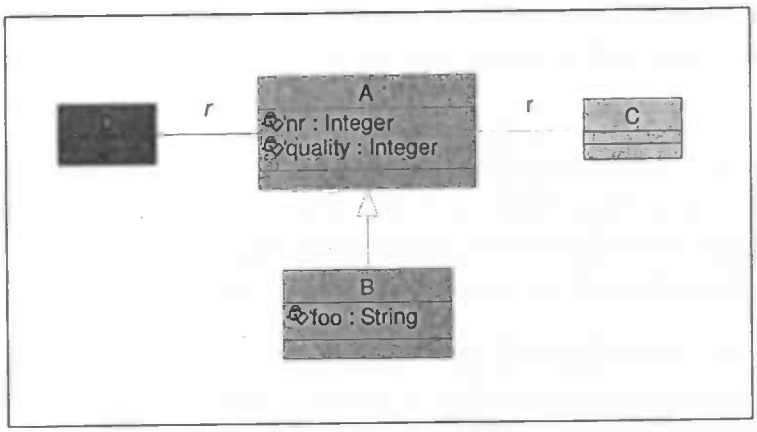

Figure 5.13: An example of the union

The data file is created in addition to the graphical result, and a screen shot of this file in a comma separated file viewer is shown in Figure 5.14. The last lines are the most important ones, they show the correspondence. When Cor1 and Cor2 are both 1, the element represents corresponding elements. When only Cor1 has a 1 the element comes from the first model and has no corresponding counterpart in the second model. For elements where only Cor2 is 1 the same holds, but then for the second model. What kind of data is put into the file can

be changed, in this case it shows the Name of the Elements, the Metaclass (type), the Stereotype and some information about the Clients and Suppliers. This could change to include everything available within the xUMLi system.

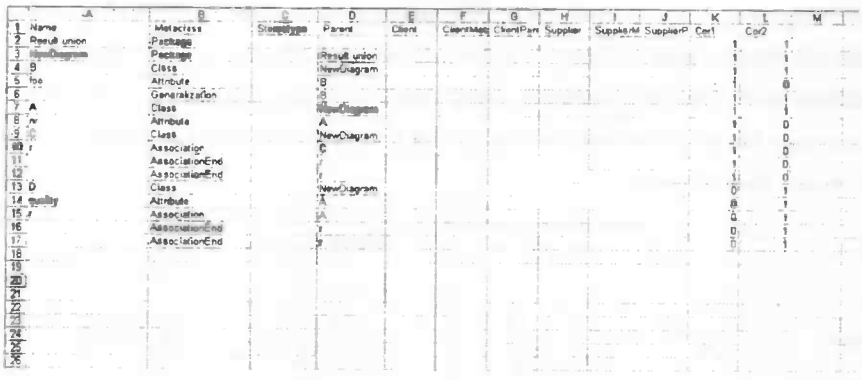

Figure 5.14: An example of the data file

In the next Chapter tests on real life industrial models are discussed.

6 Validation

6.1 Description of the used testing models

There are models of two systems available for the test cases. Both systems are large mobile phone platforms. The names of the models and systems are changed from the real system. The first system, TS, consisted of a model repository file, with models of three subsystems of the complete system. They are addressed in the following way: TS-P, TS-W and TS-M. The three parts each contain two versions of that specific subsystem. The different versions of the system are numbered: TS-P1 represents Test System functionality P, version 1. One version is specified in one diagram. These diagrams are used for the test cases. The diagrams are quite small, and contained around 25 Classifiers, 30 Dependencies (of which some Abstractions), divided among 10 packages. All the models from the TS system are forward-engineered.

The second system is will be referred to as ISA. It consists of three different model repository files. Two models are reversed-engineered, one is forward engineered. They are addressed as: ISA-R1, ISA-R2 and ISA-F. The two reverse engineered models differ slightly. They can be considered quite big(see table 6.2). IS-F is around the same size but the organization is somewhat different from the ISA-R models. The ISA system also has two sets of traces, which were created by monitoring the functioning of the system. These traces are addressed as ISA-T1 and ISA-T2.

The tests have been conducted on different computers. Sometimes the computer was doing other things at the same time (either engaged by the user or by the OS used). This is why in the testing results no statistical data about the performance is given. Note that the performance is also influenced by the used options of the set operations (the correspondence calculations method used, creating a data file, creating diagrams, etc.).

During the tests, generally three different types of change can be detected: removing, adding and moving of model elements. The first two, the removal and addition of model elements, can be enquired directly from the results, either visual (in Rational Rose), or statistic (in a data file). The moving of model elements is visible by the combination of a removing and adding the same model element. The actual detection of the moved elements will be guesswork if very detailed information about the model and its design process and history is not present. Moving could be automatically detected when comparing the differences with the parent rule to the differences without the parent rule, so with a combination of different tuned set operations. This automatic moving detection is not used for the below stated test cases.

The set operations have been used with different options; these are described in every case. Because the

models used in the test cases have also been used in the development phase, the options are already part of the current implementation of the set operations. When using new models which have completely new demands, it is possible that the set operations need to be adjusted, e.g. that rules need to be modified or added.

In the problem statement (Chapter 3) three usage scenarios were stated where the use of set operations is desired. With the scenarios some test cases have been stated. These test cases are described and explained below, in the context of the above described test models. The test results and the evaluation of the test cases will follow in the coming sections.

6.2 Testing results

6.2.1 Scenario 1: Merging of models

The merging of models is validated in two test cases.

Test case 1: merging of two different versions of a model

In this test case two forward engineered models from similar background with much communality are merged together. They describe the same functionality, but different versions. The forward engineered diagrams used for this case are TS-P1 and TS-P2. The two different versions were unified with visualization of the corresponding and differing parts. Diagram TS-P1 represents an older version of the same system as diagram TS-P2. The versions were quite close together, so most of the elements of the diagram were overlapping. The calculation time of the set operations was negligible. Two fragments of the model are given in Figure 6.1 and 6.2.

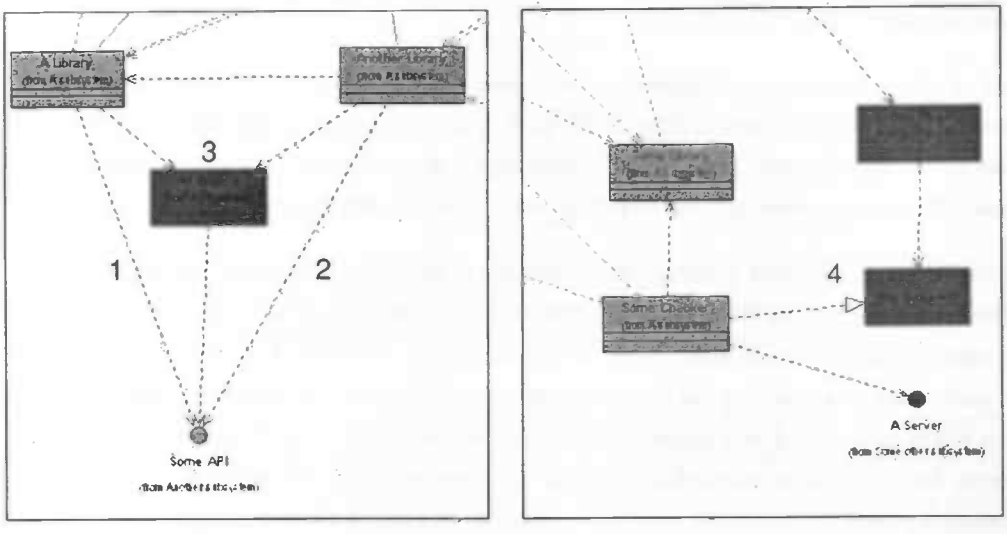

Figure 6.1 and 6.2: Two examples from the merged TS-P1 and TS-P2 models

Figure 6.1 shows a typical situation of addition of elements. The two Dependencies marked as 1 and 2 are from TS-P1, the three connected to the Class *An Adaptor* as well as the Class self are from TS-P2. In the newer model the communication to *Some API* is handled by *An Adapter* (3). This involves clearly some abstraction changes from the previous model. Figure 6-2 shows the adding of some new functionality. Two new Classes are added *A new Plug-in* and *A new API*, as well as Dependencies between them and in connection to the corresponding part of the model. One of the new Classes is a Realization of an existing Class (4).

Test case 2: merging models which represent different functionality

Two forward engineered models that represent different functionality are merged together, The focus is on the detection of the common elements as well as the detection of potential conflicts between the models. The resulting merged model should be able to express the same as the two models separately. For this test the latest versions of the TS model will be used (TS-P2, TS-W2 and TS-M2). All the three diagrams represent different functionality. Several commonalities have been found:

Used diagrams	Packages	Classes	Interfaces	Dependencies
TS-P2	11	15	15	43
TS-W2	10	10	13	27
TS-M2	14	12	15	34
TS-P2 and TS-W2	7	0	5	0
TS-M2 and TS-W2	7	1	4	0
TS-P2 and TS-M2	4	0	2	0
TS-P2 and TS-M2 and TS-W2	1	0	1	0

Table 6.1: The number of common elements in the different TS models

One Package and one Interface were used in all three models. The use of common Interfaces does not imply any potential problems. These Interfaces are probably the connection to some other part of the system. No common Dependencies in all the three diagrams were found at all. This gives some indication that the functionality described by the models will not effect the same part of the system and will probably not conflict with each other.

This test clearly showed the usefulness of the set operations within the MDA initiative context. The resulting models were probably usable within the context of the further development. The used models were of the same abstraction level and the resulting union of the models was a correct usable model too.

6.2.2 Scenario 2: Comparing different versions of a system

Two test cases have been conducted to validate scenario 2.

Test case 3: Checking different reverse engineered versions of a model against each other

Two big industrial reversed engineered models are compared with the set operations, the ISA-R1 and ISA-R1 models. The information enquired from the difference and the intersection gives an insight on the differences

between the models or the differences in the reverse-engineering process. The models describe the same system, probably only the reverse engineering process was changed between the construction of them. The following table shows the number of common elements of the models.

Used models	Packages	Classes	Dependencies
ISA-R1	167	1024	14880
ISA-R2	143	1048	8331
ISA-R1 and ISA-R2	143	696	8331

Table 6.2: The number of common elements the ISA-R1 and ISA-R2 models

All the Packages have found to be corresponding. The difference in the number of Dependencies from the different models is probably caused by the reverse-engineering process; ISA-R2 is constructed at a higher level of abstraction. What is very interesting is that all the Dependencies of ISA-R2 have found to be corresponding, while not all the Classes from that model have corresponding counterparts in ISA-R1.

Test case 4: Checking a forward engineered model against its reversed counterpart
Two models of the same system are compared; one of them is the forward engineered model, one of them is the reverse engineered one. The results can give indications about how close the forward engineered model is to its implementation, and address potential inconsistencies between the models. For this test a sufficient part of the ISA-F model is compared against a part of the ISA-R1 model. Because the organization of both of the models differed, it was needed to do some pre-processing by hand. The names are corrected so they are alike (some spaces corrected and the ISA-R1 model contained a letter in front of every model element which was not there in the ISA-F model). The resulting subsets are checked against each other. 154 of the Dependencies of IS-F were Abstractions. Because the IS-R1 did not have any abstractions, they all resulted as none-corresponding. A test run with the rules adjusted to accept correspondences between Dependencies and Abstractions also resulted in no corresponding Abstractions.

Used models	Packages	Classes	Dependencies
ISA-F	14	319	856
ISA-R1	14	191	972
ISA-F and ISA-R1	14	125	0
ISA-F and ISA-R1 With stereotype	14	37	0

Table 6.3: The number of corresponding elements the ISA-R1 and ISA-F models

One run checked the models against each other with the stereotype rule taken into account and one run checked them without that rule. When these results were compared, it was easy to say which elements had different stereotypes in the IS-R1 model compared to the IS-F model. Sometimes this was caused by the use of different conventions for the different models (<HW Driver> or <HW_Driver>), but sometimes the differences could even give an initiative for detecting inconsistencies. For example some elements are marked in the IS-F model as Library, while they are marked in the IS-R1 model as an Infrastructure.

When analyzing the results, it seemed that the forward engineered model was at a different level of abstraction as the reversed-engineered. They had an overlap of a significant number of Classes, but there were no corresponding Dependencies. The used models differed half a year in time; IS-R1 was 6 months older then IS-F. This could give an indication of why the differences between them were so big. The results themselves did not give a very interesting perspective on both of the models, but it proved that the set operations can be used as an analysis tool for models. This process is discussed in the context of a software maintenance process in [Riv04]

6.2.3 Scenario 3: Finding and visualizing behavioral slices in class diagrams.

Because one set of slices was available, one test case is used to validate scenario 3.

Test case 5: : Slicing traces on a reverse engineered model

The class diagrams of the traces from the ISA model are mapped on the ISA-R2 model. The traces are put into the context of the model and the impact of the functionality described by the traces can be observed within the hierarchy of the ISA-R2 model. The traces were enquired by the actual monitoring of the running ISA system. The trace ISA-T contained about 120.000 messages. These messages have been converted into behavioral diagrams, and these behavioral diagrams have been transferred into class diagrams. This resulting class diagram contained 64 classes and 156 dependencies. Complication was that these elements were not in the context of the reverse-engineered model (no package hierarchy was available). This is why an adjusted set operation was used, the rules were relaxed that the parent rule and the hierarchy was not taken in to account. Because the context of the packages could not be taken into account, all the elements needed to be compared. This caused an increased execution time of the set operations.

Used models	Packages	Classes	Dependencies
ISA-T	1	64	156
ISA-R2	143	1048	8331
ISA-R2 and ISA-T	0	46	48

Table 6.4: The number of corresponding elements the ISA-R2 and ISA-T models

Interesting aspect of this test run was that certain model elements were not found at all in the IS-R1 model. They turned out to be caused by an interpretation error in the pre-processing of the traces to the class diagrams. In this situation the results of the set operations directly corrected the model construction process.

6.3 Evaluation of the set operations implementation

6.3.1 The xUMLi system

The xUMLi system has proven to be very useful. The direct reflection of the UML metamodel on the xUMLi system is easily usable to construct UML processing operations. Also the possibility to use Python to construct the components is fount to be valuable. However when using the xUMLi system very intensively as a data-structure the efficiency was becoming an issue. When the corresponding information was stored in xUMLi the program got too slow.

The xUMLi system is developed to be a component based model manipulation system. It is able to combine various manipulation components to create the complete desired functionality. This works very well, only the nature of the used components demands them to be adjustable. When every component needs a control data stream to adjust them, having complex model manipulations by combining a lot of the available components becomes very complicated.

6.3.2 The set operations implementation

The nature of the different models was causing new demands for the set operations. This is why new options have been implemented, and even different versions have been made for very specific situations. The set operations component is a potentially very big component. One thing which should be done is the splitting of functionality. The set operations component has already three distinct parts: the correspondence calculation, the correspondence rules and the result management. These three parts could be split, and especially the result management should be split from the main program because this is different in nature from the other two. Problem with that is the handover of the correspondence data, which in the current implementation is a Python hash table. This table cannot be passed between the components because the Hashvalues could change.

The constructing of the correspondence rules was not very complicated. The challenge was to construct it in such a way that it was adjustable. Because the implemented rules are so concrete, it was very teasing to hardcode them in the set operations. Also the order in which the elements are evaluated was something which should be adjustable and therefore turned into a challenge. In the current implementation the order of evaluation can be set in the rules file, as a suggestion to further implementation these should also be adjustable in the control data.

The constructing of the results was more complicated. The reconnection of the Relationships and the ownerships was sometimes difficult, but solvable. It was more complex to minimize the information lost. For example, if two Classes are only different because they have different Stereotypes. They are considered non-corresponding, and they have to be outputted as non-corresponding elements. Within xUMLi this works fine, but then exporting to Rose, two elements with the same name cannot exist in the same namespace. This

caused one of the Classes to get lost.

It happened more than once that a strange output in a testing situation was not caused by the set operations, or by xUMLi, but by the input files used. The existence of "invisible" Dependencies, deleted Classes, slightly different names etc. has caused many frustrating moments. Usually it turned out that the generated output was correct, but the used input was not clear.

Interesting use for the set operations will be the combination of the operations to construct more complex behavior. For example the moving of Classes within different versions of a model can be detected when combining two differently tuned set operations. The first one will compare the models without the knowledge of the context, the second one will compare the same models with the context rules turned on. The Classes which are corresponding in the context-free result, but are not corresponding in the context-sensitive result have been moved. This comparing can be done again by the set operations.

6.4 Thesis questions

The questions stated in the problem description will be answered in this section. Every question has a small discussion and refers to the implementation and the test cases.

Can the set operations for class diagrams be implemented in an efficient, scalable, and usable way?
The tests have proven clearly that the set operations function correctly. Also they have proven to work on big industrial models. They are flexible and adjustable to different situations, as the user requires. For the above mentioned cases the set operations have been used with and without stereotype check, without using the package hierarchy, with adjusted name-rules, etc. These options are easy to set in the current implementation. Also new rules can be added easily

The efficiency depends highly on the user demands. When currently using the set operations the importing and exporting of the models takes more time than the actual calculation of the set operations, so within xUMLi the efficiency is sufficient. When the user demands a data file of the results, or is not able to use the hierarchy of the packages, the efficiency decreases. But taken the situations where the operations will be used, in reverse engineering or within MDA, there is always a big amount of processing time needed to process the models, and the set operations will not be the bottleneck in these situations.

What kind of user influence is required for the set operations?
There are potentially two different kinds of user influence for the current implementation. The first one is the "basic" user of the operations. He selects the models to be compared, what kind of rules need to be used and how he wants to see the results. For this purpose the control stream can be used, or the user can select the options in the GUI. The second way of using the set operations is by an "expert" user. This user can adjust the set operations for a specific set of models, for a specific situation or whatever is needed. He can add, remove or adjust the comparing rules easily in the rules file of the set operations. Also he can introduce new ways

of representing the result or modify an excising way. The user need not know the whole working of the set operations, but can fine-tune it by changing some parts.

Is it possible to create a general implementation which can be used without knowledge of the context of the models?

As the test cases have shown, this implementation of the set operation needs to be fine-tuned with knowledge about the nature of the models which are going to be used. The models which have been tested were quite different. This led to the situation that for the different scenarios the set operations needed to be adjusted. The most demanding adjustments are now part of the implementation, but one never knows what new situations will demand. When the UML models are all used in a universal way, the set operations will probably be universal too. But for now manual analysis is needed before using different models.

How can the set operations be exploited for the different types of UML processing operations (merging, checking and slicing)?

As test case 1 and 2 have clearly shown, the *merging* of models is very well possible with the current implementation. *Checking* the different versions of the system was clearly demonstrated in the test cases 3 and 4. Models can be checked, and the numerical data enquired from it can be used to analyze the differences between the models. The *slicing* of the models was shown in test case 5. It showed that it is possible to get an understanding of the impact of a slice on the whole model.

This implementation of the set operations have already proven itself as a model processing tool. In [Riv04] a method for software maintenance is shown, in which the set operations are used. Because the set operations will be made part of the xUMLi system, they will be used more in the future.

What is a useful way of reporting or visualizing the data enquired from set operations?

Visualization with colors has proven to be very valuable for set operations on diagrams. This is because it is a very easy way to visualize the changes and additions, especially when it is used in combination with the union operation. Both of the diagrams are then visible in the same result, and the differences are visible through the coloring. However, when the models are bigger and are not represented in one diagram anymore it is hard to visualize the results with colouring. At last some of the elements cannot be coloured (for example features), so for them a different way of reporting the differences has to be found. The use of data files can give the appropriate insight in the results, as is proven in test cases 3 and 4.

7 Conclusions

7.1 Conclusions

This thesis described a way of comparing models. The need for processing models was solved by an implementation of set operations. The set operations for class diagrams are concretely specified and implemented in this thesis. It is a flexible, adjustable and scalable implementation which is proven to work on large scale industrial models. It is used within the model processing platform xUMLi, which can potentially use models from every format, making the implementation CASE-tool independent. Within the xUMLi system the set operations can be combined with other model processing components.

Compared to other implementations of set operations or model comparing, this implementation takes into account more than a name or identifier comparison to determine the correspondence. The necessary structure of the model is checked in addition to a name, by comparing the parent and the mandatory neighbours. Also the Stereotype is checked, and the rules system makes it possible to add any other type of criterion for correspondence. Within a standard development process the set operations can be fine-tuned easily, and used through the GUI. The flexible result management module gives the user a lot of freedom to generate different kinds of results.

The set operations give the designers a tool to process models. It can automatically merge models together, and check different models for potential inconsistencies. Also it can be used to slice certain information from a model, like traces. This functionality has been shown in concrete test cases. The system is designed for, and tested with big real-life industrial models. The implementation is constructed in an adjustable way. Several options can be set, and it is very easy to add or modify functionality.

The using of coloring as a way to visualize the result has proven to be very valuable. Together with the statistical data file which is generated the designers have a good instrument to analyze their models, or their model design process. The implementation is already used within the reverse-engineering [Riv04], and will be used more within the xUMLi system [Riv04b].

7.2 Acknowledgments

The author would like to thank the following persons for the following reasons:

- Petri Selonen, for his constructive feedback and unlimited ideas, and for making me fear UML.
- Kai Koskimies, for creating the opportunity for me to go to Finland, and for his supervision in Tampere.
- Jan Bosch, for his flexible guidance of my project.
- Iris, for her long-distance support.
- Mark, for the enormous quantities of bakkies we drank together.
- Jaco, for showing me that the 3rd world is also civilized.
- Jur en Petra, for visiting me in Tamperes darkest times.
- George W. Bush, for dividing the world into good and bad again.

References

[Ala03] Marcus Alanen, Ivan Porres: *Difference and Union of Models.* In proceedings of the UML 2003 Conference, October 20 - 24, 2003, San Francisco, California, USA. LNCS 2863. Springer.

[Art03] The Art project group: *http://practise.cs.tut.fi/art*, April 2004

[Bjö00] Morgan Björkander: *Graphical Programming Using UML and SDL.* IEEE Computer 12 (33), December 2000, p. 30-35.

[Bom97] Boman M., Bubenkom J.A., Johannesson P. and Wangler B.: *Conceptual Modelling.* Prentice Hall PTR, 1997.

[Bos00] Jan Bosch: *Design and Use of Software Architectures.* Pearson Education Limited, 2000.

[Boo91] Grady Booch: *Object Oriented Analyses and Design with applications*, 1st ed. Addison-Wesley, 1994.

[Boo98] Grady Booch, James Rumbaugh and Ivar Jacobson: *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Fow00] Fowler M.: *UML Distilled Second Edition.* Addison-Wesley, 2000.

[Jac91] Ivar Jacobsen: *Object-oriented software engineering.* ACM Press, 1991

[Jac99] Ivar Jacobsen, Grady Booch, James Rumbaugh: *The Unified Software Development Process.* Addison-Wesley, 1999.

[Ken03] Kennedy Carter, *executable UML*, http://www.kc.com/MDA/xuml.html , January 2004.

[Kent78] Kent W.: *Data and reality.* Elsevier Science Inc, 1978.

[Kol02] Kollman, R., Selonen, P., Stroulia, E., Systä, T., Zündorf, A.: *A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering.* In Proc. Of WCRE'02, Richmond, Virginia, USA (2002)

[OMG02] Object Management Group: *Meta Object Facility (MOF) Specification v1.4.* http://www.omg.org/technology/documents/formal/mof.htm , December 2003

[OMG03] Object Management Group: *OMG Unified Modeling Language Specification v1.5 (2003).* http://www.omg.org/uml , November 2003

[OMG03b] Object Management Group: *OMG Model Driven Architecture.* http://www.omg.org/mda/ , December 2003

[OMG03c] *MDA Guide v. 1.0.1.* http://www.omg.org/docs/omg/03-06-01.pdf , February 2004.

[OMG04] Object Management Group: *OMG Unified Modeling Language 2.0 Infrastructure Specification.* http://www.omg.org/uml , January 2004

[Ohs03] Dirk Ohst, Michael Welle, Udo Kelter. *Differences between Versions of UML Diagrams.* In proceedings of the 9th European software engineering conference 2003, Helsinki, Finland. ACM Press.

[Pel00] Jari Peltonen, Petri Selonen, *Processing UML Models with Visual Scripts.* IN proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments, Stresa, Italy.

[PRA03] *PRACTISE research group*, http://practise.cs.tut.fi/ , May 2004

[Pyth] *Python*, http://www.activestate.com/Products/ActivePython/ , May 2004

[Rat04] Rational Software: *Rational Rose*, http://www.rational.com , May 2004

[Riv04] Claudio Riva, Petri Selonen, Tarja Systä, Jianli Xu, *UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance*. unpublished manuscripts, submitted.

[Riv04b] Claudio Riva, Petri Selonen, Tarja Systä, Antti-Pekka Tuovinen, Jianli Xu, Yaojin Yang: *Establishing a Software Architecting Environment*. unpublished manuscript, submitted.

[Rum91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991

[Sel03] Petri Selonen: *Set Operations for Unified Modeling Language*. Proceedings of the Eighth Symposium on Programming Languages and Software Tools, 17-18 June 2003, Huopio, Finland.

[Sel03b] Petri Selonen, Kai Koskimies, Markku Sakkinen: *Transformation between UML Diagrams*, Journal of Database Management, 2003

[Sel04] P. Selonen: *Phd thesis of Selonen*, Phd thesis, Technical University of Tampere, Tampere, Finland, 2004. Unpublished manuscript.

[Sii02] M. Siikarla, J. Peltonen, and P. Selonen: *Combining OCL and Programming Languages for UML Model Processing*, In Electric Notes in Theoretical Computer Science (ENTCS) dedicated to the UML 2003 workshops, Elsevier publishing, San Fransisco, CA, USA, October 2003. Accepted for publication. Preliminary version available on-line at http://i11www.ira.uka.de/ baar/oclworkshopUml03/

[Som98] Ian Sommerville: *Software Engineering fifth edition*. Addison-Wesley, 1998

[xUMLi] Petri Selonen, Jari Peltonen: *An Approach and a Platform for Building UML Model Processing Tools*. Unpublished manuscript, submitted, 2004.