JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Reasoning over UML Models with Ambiguities

MASTER'S THESIS

submitted in fulfilment of the requirements for the academic degree

Diplom-Ingenieur

in the master study of

SOFTWARE ENGINEERING

submitted by:
Franziska Öllerer B. Sc.

completed at:
Institute for Systems Engineering and Automation

referee:
Univ.-Prof. Dr. Alexander Egyed M. Sc.

supervisor:
Univ.-Prof. Dr. Alexander Egyed M. Sc.

Linz, January 17th, 2013

Abstract

A software designer has the ability to define a system using the Unified Modeling Language (UML). However, a concept to document design decisions along the modelling process or to add and try different decisions is yet to be developed.

This thesis introduces the Ambiguity Concept. This concept enables its user to add different design decisions, so called *ambiguities*, to design models. In addition, this thesis shows how a consistency checker can be employed to detect inconsistencies in a model with ambiguities.

In addidion, the *Ambiguitymanager*, a plugin for the IBM Rational Software Architect is developed. The *Ambiguitymanager* can be used to add ambiguities to UML models and to conduct reasoning steps. Although all examples are realised with the UML, the Ambiguity Concept is applicable for any kind of modelling languages as long as they are based on a well-defined meta-model.

Implementing a large case study in assistance with the *Ambiguitymanager* proved that the Ambiguity Concept is applicable to a huge number of model elements. The case study shows that the *Ambiguitymanager* offers the possibility to define optional elements instead of re-modelling entire model parts depending on a decision. Furthermore, with the *Ambiguitymanager* users can select particular model elements of a complex model and add certain ambiguities instead of re-modelling the whole diagram. In addition to the case study, the *Ambiguitymanager* and the Ambiguity Concept are evaluated in terms of perceptions resulting from the implementation of the case study.

Zusammenfassung

Mit der Unified Modeling Language (UML) ist es einem Designer möglich, ein Software-System zu spezifizieren. Auch wenn diese Sprache ein weites Spektrum an Sprachkonzepten definiert, so gibt es jedoch noch kein Konzept, um ausgewählte und auch ausselektierte Designentscheidungen aufrecht zu erhalten und miteinander zu kombinieren, bzw. auszuprobieren.

Diese Thesis stellt das Ambiguity Concept vor. Mit diesem Konzept ist es möglich, unterschiedliche Designentscheidungen, sogenannte *ambiguities*, in Entwurfsmodellen zu definieren. Des Weiteren wird gezeigt, wie ein *consistency checker* verwendet wird, um Inkonsistenzen in Modeldefinitionen mit *ambiguities* aufzudecken.

Außerdem wird ein Plugin namens *Ambiguitymanager* für den IBM Rational Software Architect entwickelt. Mit diesem Plugin kann ein Designer *ambiguities* in UML Modellen definieren, verwalten und Konsistenzprüfungen durchführen. Auch wenn alle Beispiele in dieser Arbeit durch die UML beschrieben sind, ist das Ambiguity Concept auf jede andere Art von Modellierungssprache mit einem definierten Metamodell anwendbar.

Um zu zeigen, dass das Ambiguity Concept ebenso auf Modelle mit einer großen Anzahl von Modellelementen anwendbar ist, wird eine Fallstudie mit dem *Ambiguitymanager* implementiert. Durch diese Implementierung wird deutlich, wie ein Designer optionale Modellelemente definieren und somit Modellierungsarbeit einsparen kann. Abschließend werden die aus der Implementierung der Fallstudie gewonnen Aspekte in Bezug auf den *Ambiguitymananger* und das Ambiguity Concept diskutiert.

Acknowledgements

I would like to thank Prof. Dr. Alexander Egyed for guiding me through this thesis. Our discussions helped me understand the complexity of modelling aspects. Additionally, I thank Dr. Roberto Erick Lopez-Herrejon for suggesting such an appropriate case study.

Furthermore, I thank Prof. Dr. Klaus Schmid for his support and advices in terms of research literature for Software Product Lines.

I would like to thank DI Philipp Lengauer for his support through my master studies and his knowledge about formalism and verification. Additionally, I thank DI Markus Koppensteiner for helping me understanding networks and I thank Daniel Neidhart for his PowerCollection class. Sebastian Schubert, thank you for the dinner every Sunday.

Anna Deckmann, thank you for your helpful tips concerning the English language.

Sincere thanks to my family for their steady support and encouragement.

Finally, I would like to thank Dirk Bahl for his patience and love.

Index of Contents

1 Introduction

Imagination is more important than knowledge, because knowledge is limited.

(Albert Einstein)

In the design phase of a software system, stakeholders are confronted with a huge number of ideas and design decisions. While some of them might be adequate for the architecture of the desired system, some might be dissatisfactory and will be discarded. This masters' thesis deals with the expression of different design decisions, so called ambiguities, in design models. The Ambiguity Concept introduced in this thesis provides a possibility to express different design decisions in terms of any kind of models and model elements. The Unified Modeling Language (UML) [1] is the de facto standard modelling language that is used to design a software system concerning its behavioural and structural aspects. Due to this, the Ambiguity Concept is discussed in terms of the UML.

1.1 Motivation

Although a software designer has the ability to define a system with the UML, a concept to document design decisions along the modelling process has yet to be developed. While one might say that discarded design decisions are dissatisfactory and have been excluded for a reason, they might prove valuable later on if requirements are changing or if previously made decisions clash with general requirements or basic conditions. In this case, rejected decisions may have to be reviewed again.

Furthermore, an assistant that tells the designer, which design decisions may cause inconsistencies in the model based on defined constraints and which satisfy a particular need, does not yet exist. Thus, this thesis discusses ambiguous decisions of design models and how they can be evaluated in terms of defined conditions.

Software product lines can be considered as a particular paradigm that is concerned with handling different decisions in terms of variability of derived products. There are several approaches e.g. Gomaa's [2] and the general approach of commonality and variability modelling in software product lines. The latter focuses on a component-oriented approach such as features encapsulating a particular functionality of the system. However, Gomaa also focuses on feature models in terms of software product lines, but his PLUS (Product Line UML-Based Software Engineering) approach provides an extension to the existing UML in terms of introducing concepts to depict optional, mandatory, and alternative aspects.

Product lines can contain mutually exclusive features, never to be selected together in a configuration for a derived product. If for some reasons such features are selected together, the derived product can be considered as inconsistent due to the fact that its configuration violates defined rules between features.

Consistency checkers can help an engineer find a consistent configuration. They detect inconsistencies with respect to defined feature models and their optional and mandatory features. Even though approaches to express variability in product lines exist, a concept to express such a decision for any kind of model is still missing. Moreover, there exists no solution enabling its user to handle different design decisions in terms of their compatibility or to find the best combinations of decisions for a defined problem.

The Ambiguity Concept aims at providing an approach to document different design decisions by keeping them alive even if they are not selected for a particular design model. The concept also provides a mechanism that can be used to find all combinations of defined design decisions if they depend on one another.

In addition, it can be used to try different combinations of design decisions. In the course of this thesis, a tool implementing the Ambiguity Concept has been developed. This tool is called *Ambiguitymanager* and provides the ability to add ambiguities to design models. In addition, the *Ambiguitymanager* is linked to an existing model analyser tool. Combined they can be used to conduct consistency checks, i.e. checking if the model is still valid with different combinations of design decisions. Thus, the tool provides the detection of inconsistencies in model definitions.

1.2 Goal

The goal of this thesis is to define a language to express different design decisions for design models, so called ambiguities. Furthermore, a tool that realises this approach by providing the management of ambiguities for UML model elements is implemented. Additionally, the implemented tool is connected to an existing model analyser that provides consistency checking of combined design decisions. Such a decision is called a *choice*. A *choice* results from ambiguities and existing mandatory values of a model element and a depending property for which the ambiguity was created. The consistency checker performs reasoning steps and takes *choices* as input. Furthermore, it combines *choices* if they affect one another and returns so called *determinations*, which can be declared as consistent or inconsistent.

Hence, the reasoning process can identify a set of *choices*, which can never lead to a consistent model. Consequently, the consistency checker informs a designer about whether his design decisions are defined in an inconsistent way.

To show that the Ambiguity Concept can be used with a huge number of model elements, a large case study is implemented in association with the *Ambiguitymanager*.

In addition, the *Ambiguitymanager* and the Ambiguity Concept are evaluated in terms of perceptions resulting from the implementation of the case study.

1.3 Scope of Work

This thesis focuses on expressing unsolved design decisions of model elements defined with a software modelling language. The Ambiguity Concept can be adapted to any kind of software modelling language if there exists a well-defined meta-model. However, this thesis takes the UML as a representative modelling language due to its widespread popularity.

1.4 Terms and Definitions

The terms user, designer, or engineer describe a person who defines a system design and interacts with particular software tools. All those terms can be considered as equals.

Furthermore, the terms design decision, decision, or choice represent a possibility to model a certain aspect of a system and can be considered as equals. A *choice* printed in cursive characters denotes a unique *choice* of the Ambiguity Concept.

Finally, a model element or element denotes a particular model aspect or model instance as defined in the UML.

1.5 Structure of the Thesis

Chapter 2 provides an overview about the basics as well as background knowledge needed to understand important aspects of adding different design decisions to models. On account of this, the Unified Modeling Language and consistency checking of models will be pointed out. Additionally, background aspects of adding different design decisions to UML models will be discussed. The chapter ends with an overview about related work, such as Product Line Engineering and model analysers.

Chapter 3 presents an approach to the so called the Ambiguity Concept and illustrates its usefulness in terms of adding different design decisions, called ambiguities in the context of this thesis, for UML models. Additionally, the reasoning process over models with ambiguities will be illustrated.

Chapter 4 deals with the implementation of the *Ambiguitymanager*, a plugin for the IBM Rational Software Architect. It will be explained how this plugin can be used to add ambiguities to UML models and to conduct reasoning steps.

Chapter 5 describes the implementation of a huge case study in terms of the Ambiguity Concept. After providing concrete examples of that implementation, problems concerning the conducted reasoning process will be discussed.

Chapter 6 deals with problems and limitations of the Ambiguity Concept and the *Ambiguitymanager*. Additionally, it offers concrete examples and possible solutions to problems one might encounter.

Chapter 7 is the summary and conclusion of the thesis. In addition, it provides an overview about possible further work in terms of strengths and weaknesses of the Ambiguity Concept, the *Ambiguitymanager,* and the conducted reasoning processes.

2 Basic Knowledge

This chapter provides an overview about the basics as well as background knowledge required to understand important aspects of adding different design decisions to models. The Unified Modeling Language (UML) [1] and its diagrams are introduced, some examples of practical usage provide illustration of the processes. Furthermore, the question of how to check the consistency of UML models and what mechanisms and approaches exist will be discussed. Additionally, some background information regarding the process of adding different design decisions to UML models will be provided. Furthermore, Ambiguous Reasoning, first published by Egyed et al. [3], will be discussed; technological aspects as well as the weakness of their approach will be pointed out. Finally, an overview about related work such as Product Line Engineering and model analyser will be provided.

2.1 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the most common software modelling language for object-orientated systems and was invented by Ivar Jacobsen, Grady Booch and James Rumbaugh in 1997. Later, the UML was offered to the Object Management Group (OMG) for standardization. On November 14, 1997 the OMG adopted the UML 1.1. Since then, the UML is a standard modelling language for object-orientated systems. [1] The current specification of the superstructure of the UML version 2.4.1[1] was established in August 2012 and consists of 732 pages. The document defines structural and behavioural meta-models of UML elements and relations among them.

A modelling language can help to improve communication between stakeholders in a development process. Additionally, the capability to visualize a problem and to discuss design decisions in an early phase of the development process may avoid further mistakes. Thus, a standardized modelling language with a defined vocabulary and rules how to combine its words is essential in a software development process.

[111] The UML is property of the Object Management Group and is available at http://www.omg.org/spec/UML/2.4.1/ (2012-02-05).

The UML is a language to visualize, specify, construct, and document a software system along its development process. [1] To point out the power of the UML, a summary of the range of application will be provided.

The UML version 2.4.1 consists of 14 diagrams employed to define structural and behavioural properties of a system. Figure 1 [1] outlines the hierarchy and structure of its 14 diagrams. The modelling language contains of 7 structural diagrams, e.g. class and object diagrams, to describe system components and relations between them. To define behavioural properties between system components the UML also comprises 7 diagrams, such as sequence and state machine diagrams.

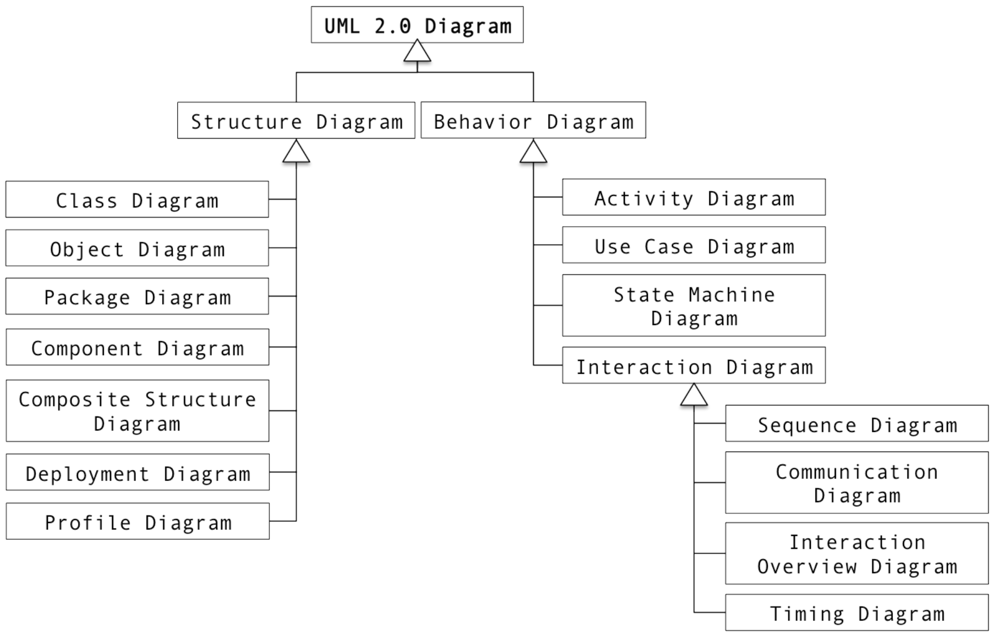


Figure 1:_ Hierarchy and structure of UML diagrams [1].

Some UML examples are essential to ensure a common comprehension of the UML. In the scope of this thesis, the used diagrams are class, sequence, and state machine diagrams. Figure 2 shows an UML class diagram of a simplified mp3 player scenario. A class is represented by a rectangle with compartments for class operations and class attributes.

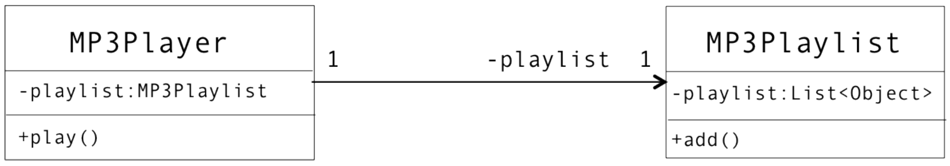


Figure 2: UML class diagram of a fictive mp3 player scenario.

As illustrated by the diagram, the scenario consists of two classes. The left class is named MP3Player and has the attribute mp3Playlist as well as an operation called play. The right class is labelled MP3Playlist and has the attribute named playlist as well as an operation called add. The plus and minus chars symbolise the visibility (+ = public, - = private) of attributes and operations.

The relation of these two classes is described by an association and represented by a line between them. The arrow in the direction of the MP3Playlist class signifies that an object of MP3Player uses or imports functionalities of an object of MP3Playlist. In this case, MP3Player has an attribute mp3Playlist with the type MP3Playlist. A number or an asterisk describes the multiplicity of an association for zero to infinity; they are often found on each end of the line. It is also possible to add roles for an association. This is illustrated by a role name placed next to the class. In this scenario, one can read the association as follows:

1. The MP3Player has only one MP3Playlist and has no certain role.

2. The MP3Playlist belongs to only one MP3Player and has the role playlist.

Figure 3 shows an UML sequence diagram of the mp3 player scenario. The two rectangle objects are called lifelines. The left lifeline is labelled with c and has type MP3Player, the right lifeline is labelled with p and has the type MP3Playlist. The arrow from c to p signifies that c calls p with an operation called add. A vertical rectangle symbolizes active parts of a lifeline object.

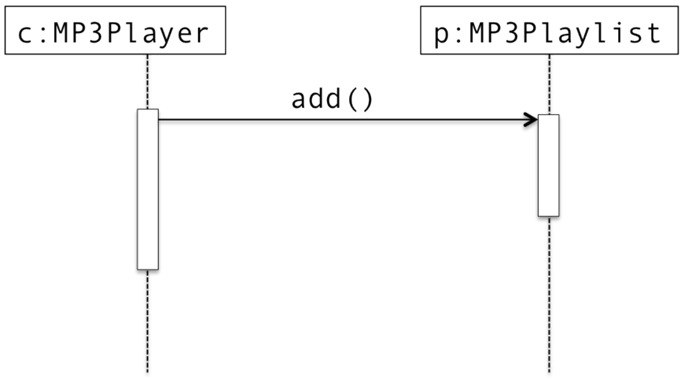


Figure 3: UML sequence diagram of a fictive mp3 player scenario.

The last required UML diagram is the state machine diagram. It is depicted in Figure 4. This diagram type is used to describe behavioural properties of system components, respectively components' states. A state machine describes possible states of a component by a given sequence of actions, states and reactions. [4]

Figure 4 shows a simplified state machine of the MP3Player and its states. The filled cycle on the left signifies the start of the sequence. A rounded rectangle denotes a state of the MP3Player. A transition between two states is symbolised by an arrow, the performed action changes into the next state. The filled cycle with a border on the far right stands for the end of the state sequence.

As depicted in Figure 4, the MP3Player's state chart diagram consists of two states *On* and *Playing* and of two transitions *play* and *off*. The sequence starts with the *On* state. If e.g. a user presses the play button on his mp3 player, the MP3Player class changes into its *Playing* state.

In reality there may be more possibilities left to decide, e.g. if the user chooses to play a certain song or stops, pauses or continues the playback. However, in this simplified scenario the only option is to continue to the end by pressing e.g. the off button.

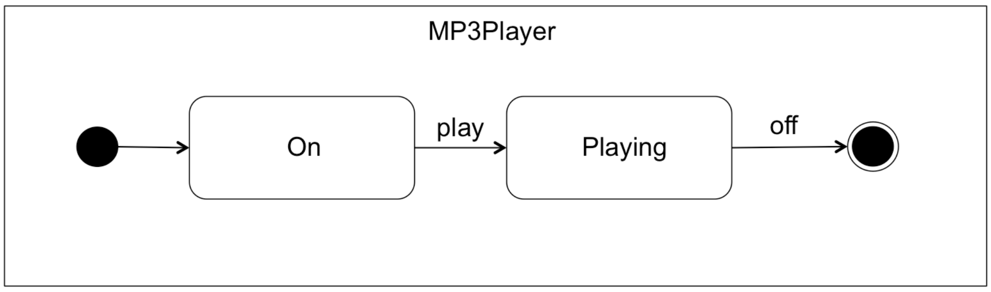


Figure 4: UML state machine diagram for the MP3Player.

2.2 Consistency Checking in UML

The immediate detection of inconsistencies in design models can save the designer redundant work. [5] The following paragraph provides an overview about the process of detecting inconsistencies in UML models.

The UML defines a meta-model for loosely coupled diagrams and design notations. With the UML, designers can define a system's structure and behaviour from different viewpoints. Inter-relations between different UML views are not defined adequately in the meta-model. Thus, the meta-model lacks sufficient declaration to define rules, dependencies, and semantic aspects among UML diagrams. [6], [5]

With the UML, designers can construct systems by using the divide and conquer strategy. This means that de facto UML diagrams benefits designers in allowing them to divide design concerns into different viewpoints by applying separating aspects. Additionally, this division can reduce complexity. Since smaller pieces of a system's properties are easier to understand, it is useful to focus on partial aspects.

A problem with separating aspects into different viewpoints and diagram-centric approaches is that, in fact, diagrams and viewpoints are not independent and do affect one another. In addition, different viewpoints of the same system can cause redundancies in terms of model information. Thus, mechanisms and methods are required to check the consistency of all redundant aspects. Such mechanisms are confronted with semantic and language variations, complicating the task of consistency checking. To check the consistency of diagrams, common consistency rules and assumption must exist and be defined in a consistent manner. [7]

Hence, two approaches are used to check the consistency of models. The consistent transformation approach transforms source models into target models by using well-defined transformation steps, which should guarantee consistency. The consistency comparison approach ensures consistency by a well-defined comparison of source models and target models to detect inconsistencies. [8]

The following sections deal with rules that can be used to define conditions to comply with during an evaluation. Furthermore, an overview about model analysing approaches will be provided. Finally, Constraint Satisfaction Problems (CSPs) will be introduced.

2.2.1 Consistency Rules

Consistency rules for UML models can be used to check whether the model satisfies defined conditions. A UML model can be considered as a valid model if it satisfies those rules. [9]

The following two consistency rules in Table 1, taken from Egyed et al. [9], exemplify how such rules can be defined in a consistent manner. Each of them describes a rule for an UML sequence diagram.

The first rule states that for each name of a message in a sequence diagram between two lifelines, an operation in the receiver's class with the same name must exist. In the implementation of this rule it needs to be checked if the receiver's class contains an operation named with the name of the message.

Concerning the sequence diagram depicted in Figure 3, the name add of the message between the sender c (MP3Player) and the receiver p (MP3Playlist) must be an operation in p's class definition. Figure 2 shows the class definition; since an operation named add exists in this sequence, the diagram can be considered as valid concerning the first rule.

Rule	Condition

1	**`Name of message must match an operation in receiver's class`**
	`operations=message.receiver.base.operations`
	`return (operations->name->contains(message.name))`

2	**`Calling direction of message must match an association`**
	`in=message.receiver.base.incomingAssociations;`
	`out=message.sender.base.outgoingAssociations;`
	`return (in.intersectedWith(out)<>{})`

Table 1: Consistency rules for UML models.

The second rule seems a bit more complex but simply checks if a sending lifeline, respectively its depending class, is allowed to call the receiving lifeline. This can be validated by checking if an association between the depending classes exists. Additionally, the rule must cover whether the association direction matches the sender and receiver roles, i.e. if an incoming association in the receivers' class definition from the sending class and vice versa, but with an outgoing association of the sending class exists.

Concerning the sequence diagram depicted in Figure 3 the calling direction of the message between the sender c (MP3Player) and the receiver p (MP3Playlist) must match an association in p's and c's class definitions. Figure 2 shows the class definitions. Since an association between the MP3Player and the MP3Playlist exists and the association's direction matches the message calling direction, this sequence diagram can be considered as valid concerning the second rule.

Both examples show the dependencies between a system's components in terms of their inter-relations among their different representations and viewpoints.

2.2.2 Model Analysing Approaches

A model analyser is used to detect inconsistencies in models automatically according to given consistency rules. Benavides et al. describe in [10] different automated consistency checking approaches grouped by the employed method or logical paradigm. Furthermore, they examine those approaches in terms of software product lines and feature diagrams. Software product lines will be introduced in chapter 2.3 Related Work.

Benavides et al. separate the approaches into three groups, namely Propositional Logic based analyses, Constraint Programming based analyses, and Description Logic based analyses. This section provides an overview about these analysis methods.

The first model analysing method is named Propositional Logic. In this approach a SAT solver [11] tries to evaluate a propositional formula to true by using appropriate variable assignments. There also exists a Binary Decision Diagram (BDD) approach, where a propositional formula is translated into a graph to try different assignment statements and to count possible solutions. [10]

The second analysis approach is named Description Logic. This method consists of concepts such as classes, roles (relationships among them), and individuals. A depending reasoner supports correctness and consistency checking as well as additional reasoning operations of a problem that is expressed with a Description Logic method.

In the third model analysing method called Constraint Programming, a CSP solver tries to solve a certain Constraint Satisfaction Problem (CSP) [12]. A CSP comprises variables, their domains, and depending constraints describing restrictions of particular domain values. Thus, Constraint Programming offers heuristics and algorithms that are concerned with CSPs. [10] Propositional formulas as used in Propositional Logic approaches can only be evaluated to true or false, however, CSP solvers can handle numeric domains of variables as well. The Constraint Satisfaction Problem and related approaches to handle such problems will be discussed in the next subchapter.

This thesis focuses on model analysers working with a Constraint Programming approach, such as the *ModelAnalyzer* [5] that will be introduced in chapter 2.3 Background.

2.2.3 Constraint Satisfaction Problem

Constraint Satisfaction Problems can be classified as belonging to the area of Artificial Intelligence research. [13] A Constraint Satisfaction Problem (CSP) consists of a set of variables, a domain of possible values for the variables, and constraints, which must be fulfilled by a variable. Thus, constraints specify allowed combinations of variables and their values. Additionally, a CSP comprises assignments of values to variables, where an *inconsistent* assignment violates a given constraint and a *complete* assignment satisfies all constraints. However, if there is no possible solution, the CSP can be considered as not resolvable. [3]

Concerning UML model elements, variables can be regarded as model elements and their particular properties such as a name or a specific relation to another element. On account of that, a domain for a variable can consist of a set of possible strings for the name property; a value represents a certain string of that set. Accordingly, a constraint can be represented by a consistency rule as introduced in 2.2.1 Consistency Rules. Due to the fact that model elements can affect one another, the challenge lies in finding valid combinations, i.e. combinations fulfilling all constraints, and in returning feasible combinations of model elements and their properties.

A CSP solution technique (CST) [14] can be used to reduce invalid combinations of assignments. In software development CSTs can be used to automatically reduce the number of impossible decisions, which must be made in activities during a software development process.

CSTs use constraint propagation and domain reduction. While the first identifies invalid decisions, the latter focuses on the remaining valid decisions. Hence, constraint propagation restricts the domains to reduce the solution space. Since variables are connected with one another by constraints, the restriction of domains of particular variables causes the restriction of other domains and variables as well. Thus, the number of possible valid allocations of all variables can be increased.

Concerning this thesis, CSTs can be used to reduce the set of design decisions, which violate a design model. However, CSTs can only compute solutions, which do not violate a model in terms of given variables and constraints. They are unable to consider constraints, which cannot be formulated in a defined way. [3]

Constraint propagation is used here in terms of model elements and their defined unsolved design decisions. The technique is used to compute solutions, which comprise valid combinations of decisions for model elements and their properties with regard to dependencies and relations between them.

2.3 Background

This chapter deals with the approach of Egyed et al. [3] who first came up with the idea of adding ambiguous or different design decisions to design models. After describing their basic concept, an overview about the *ModelAnalyzer* [5], a consistency checker for UML models, will be provided.

2.3.1 Adding Ambiguous Design Decisions

Design choices occur by disagreements of stakeholders' interests or opinions and alternative implementations. [15] Certain decisions cannot be expressed formally due to the fact that they are made by personal reasons (experience, taste). [3]

Egyed et al. [3] describe an approach to support the maintaining of stakeholders' interests during a software design process. Furthermore, they show how to express design decisions in a formal way and how to reason over the model with ambiguous decisions and well-formed rules.

The authors use the software modelling tool IBM Rational Software Architect (RSA) [16], property of the IBM Rational Software [17] division. With the RSA a software designer can develop architectural and behavioural properties of a system with the UML. Hence, a software designer has the ability to choose within a huge number of representations of development artefacts offered by the RSA. [3]

RSA uses the Eclipse Modeling Framework Project (EMF) [18] that is part of the Eclipse Modeling Project (EMP) [19]. The EMP is a project of the Eclipse Project [34]. Furthermore, the EMP is concerned with the development, progression and promotion of technologies in terms of model-based development. It provides a wide range of open source modelling frameworks, standard implementations, and tools. [19]

The EMF is one of the modelling frameworks and can be used for the development of applications, which are based on structured data models. Furthermore, the EMF provides tools to create and edit models, a basic editor, and a set of Java classes. [18]

Egyed et al. developed a plugin for the RSA in order to manipulate UML elements such as classes, lifelines or class operations in terms of their UML features. A UML feature describes an UML property of an UML element. For example, a class owns the UML feature *ownedOperation* that describes its class operations.

Additionally, the EMF specifies opposite features for some features, if this is necessary in terms of affections regarding related model elements.

An example for such an ownership relation is the *class* (ownership) feature of an operation and the opposite feature in the EMF specification *ownedOperation* of a component (class, interface). De facto an operation is owned by a component.

In contrast, the *superclass* feature of a class has no opposite feature for the related superclass, due to the fact that the superclass is not directly affected or depends on the subclass like the subclass is doing on its superclass.

The authors defined multiple design possibilities for such features and validated the whole UML model with the consistency checker *ModelAnalyzer* [5], another plugin for the RSA. The algorithm to reason over models with ambiguities is validated by Egyed et al. with a huge number of third-party models and can be found in [3].

The weakness of [3] is that all different design decisions must be added manually and directly into the program code. Thus, there exists no graphical user interface. Additionally, the output of the reasoning process is not presented in a user-friendly manner and all reverse dependencies for model elements are not generated automatically. It remains impossible to add ambiguous design decisions without a distinct knowledge of the Eclipse Modeling Framework.

2.3.2 The ModelAnalyzer

Designers are confronted with a huge number of model elements during the design phase of a software system. Ensuring the consistency of the whole model can become an overwhelming task. Model elements and different representations of model aspects such as diagrams affect one another and depend on made definitions. Thus, changing a location in a model can affect depending model parts and render the model inconsistent.

On account of that, a model analysing tool can support a designer by detecting inconsistencies immediately if a model changes. In addition, it can provide a mechanism to track existing inconsistencies and affected model parts. [5]

The *ModelAnalyzer* was developed by Egyed et al. [5] and realises a consistency checker to evaluate UML models in terms of defined consistency rules. Furthermore, the *ModelAnalyzer* operates on the Constraint Programming method mentioned in 2.2.2 Model Analysing Approaches. Thus, the *ModelAnalyzer* operates as a CSP (Constraint Satisfaction Problem) solver that tries to solve a certain CSP [12]. A CSP comprises variables, their domains, and depending constraints describing restrictions of the values of variables. A CSP solver tries to find an adequate assignment of variables that fulfils all defined constraints expressed through consistency rules. Chapter 2.2.3 Constraint Satisfaction Problem deals with CSPs.

The *ModelAnalyzer* detects inconsistencies and tracks existing inconsistencies over time. By doing this, the *ModelAnalyzer* uses a consistency rule as a black-box constraint and identifies affected model elements by observing the behaviour during its evaluation. Using a consistency rule as a black-box means that there are no exact definitions of possible values needing to be fulfilled. However, it is a generic approach. A rule for an UML model de facto *provides navigation instructions* [3] to identify affected elements and properties.

The following example shows in detail how model changes can affect different model representations such as sequence and class diagrams.

The first class diagram depicted in Figure 5 shows the mp3 scenario introduced in Figure 2. However, the add operation in the MP3Playlist class has been removed. There is a new operation called *set,* that takes over the same functionality as the add operation. Due to this, changes in the class diagram are marked in yellow. Thus, if a user wants to play a particular mp3, he presses the play button on the display and the MP3Player calls the MP3Playlist with its set operation to put the selected song into its playlist.

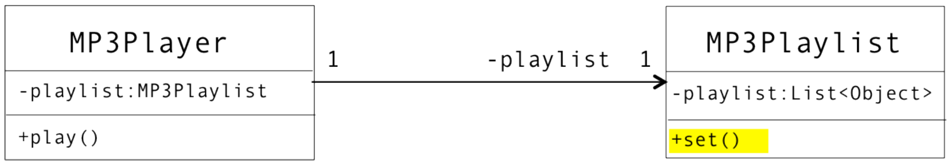


Figure 5: MP3Playlist with a new set operation.

Due to the modification in the MP3playlist's operations, a question arises: Which other model elements or view representations can be affected from that change?

Figure 3 shows the sequence diagram that realises the behaviour of the mp3 player if a particular mp3 is selected. Since the add operation in MP3Playlist class definition has been removed, the call between the MP3Player and the MP3Playlist causes an inconsistency. This is the result of a consistency rule introduced in 2.2.1 Consistency Rules. This rules states that the name of a message must match an operation in the receiver's class. The receiver here is the MP3Playlist, but there is no operation named add in its depending class definition. Thus, the *ModelAnalyzer* detects an inconsistency by observing that the rule affects the MP3Playlist class definition and that it cannot find an add operation. Figure 6 shows the visualised inconsistency based on Figure 3. The inconsistency is marked in red.

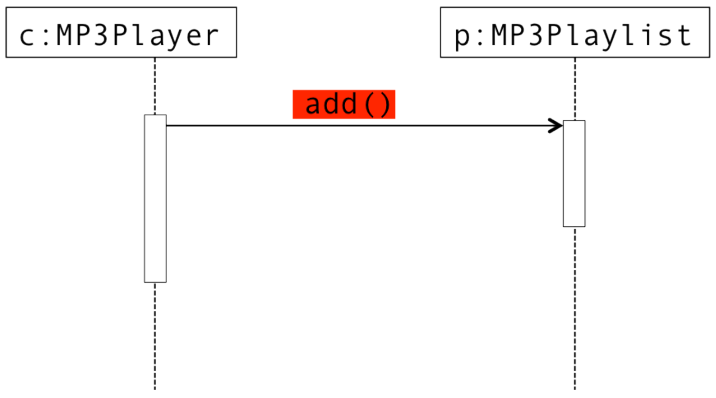


Figure 6: Inconsistency in the defined sequence diagram.

This example shows how changes in a model part can causes inconsistencies in depending parts. In the context of this thesis, the *ModelAnalyzer* is used to detect inconsistencies with a set of different and ambiguous design decisions of UML models. An in-depth explanation as to how the evaluation is conducted will be provided in 3.3 Reasoning over UML Models with Ambiguities.

2.4 Related Work

In this chapter an overview about related work will be presented. Product Line Engineering will be introduced and a modelling approach for software product lines with the UML will be discussed.

A software product line (SPL) or software family consists of a range of software instances, which are build on the same platform. Products of the platform share more commonalities than variability. [10]

Furthermore, such a platform consists of core components on which every software product that is part of the SPL, is build. The shared platform consists of architectural aspects, domain models, software components, requirement documents, test artefacts such as test plans and test cases. [20], [21] A concrete product of a product line can use additional software components, which are not part of the platform. Thus, the products derived from a SPL differ in individual configurations and additional functional or non-functional requirements.

There are a number of definitions for SPLs in the research area. In this thesis, two definitions focussing on different perspectives will be mentioned. While the first definition of Clements et al. [22] describes SPLs from a market-driven perspective, Bosch et al. [23] takes on a technological oriented point of view. [24]

In [22] a SPL is defined as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way*".

However, Bosch defines SPLs as follows: "*A SPL consists of a product line architecture and a set of reusable components designed for incorporation into the product line architecture. In addition, the PL consists of the software products developed using the mentioned reusable assets*". [23]

Product Line Engineering (PLE) is a paradigm that is concerned with the effective and efficient development of SPLs. It is used to identify common functionalities and variability in software families. Pohl et al. [25] describe the improvement of quality and the decrease of development cost and Time To Market (TTM) as the core advantages and motivations for PLE. Due to the fact, that reused software artefacts are tested in different parts of a system, the quality of artefacts can be increased. Furthermore, decreasing development costs results due to reuse of software artefacts, which can be used in more than one software product even if higher development investments must be made at the beginning. However, the initial development costs of a SPL's core artefacts (platform) are high, however, reuse of artefacts can reduce TTM for individual products. [25] Hence, the SPL paradigm can also be considered as a cost-effective approach that helps popular companies such as Motorola or Hewlett-Packard to increase their productivity and customer satisfaction related to quantitative and quality gains. [26]

PLE includes two major process steps. The first is called *Domain Engineering*. This step consists of three phases: The domain analysis phase, the domain design phase and the domain implementation phase. The domain analyse includes the identification of commonality and variability aspects of a SPL. The domain design phase consist of activities to develop core assets and the definition of the SPL architecture. During the domain implementation phase the defined architecture is implemented.

The second major step in PLE, named *Application Engineering*, describes the development of final products based on core assets and additional product-specific customer requirements. [24]

Two approaches are used to express variability in software product lines, first feature modelling (FM) and second decision modelling (DM). A comparison of both approaches can be found in Schmid et al. [27]. In chapter 5 feature models are used to express variability in the focused case. Additionally, an overview about core aspects of decision modelling will be provided.

A feature model is the most popular method to express variability of SPL products. [24] On account of that, a feature model comprises all information of possible products of a SPL and describes features and relationships among them. [10] Furthermore, a feature diagram contains functionalities or features, which are part of any derived product of the product line and thus part of the platform. There are additional features, which describe a variation point and its concrete manifestation depending on the derived products.

To understand a feature model a simplified feature diagram of a product line for mobile phones is introduced. The feature diagram used in this thesis was adapted from [27] and is depicted in Figure 7.

Products of the mobile phone product line must have an address book and functionality allowing data transfer. The address book must offer the functionality to add contact data in the form of plain text. In addition, a derived product may have the opportunity to take a photo and to add it to a particular contact. The latter is considered as optional as it is not a mandatory requirement for a product. The data transfer can be realised via UMTS or GPRS. Furthermore, a product can have a camera used to take pictures. The camera can be equipped with a zoom or a self-timer.

The example in Figure 7 depicts such a Feature Diagram of the product line for plain mobile phones. A feature model can also be expressed in a tree notation as in [28].

The first level consists of one rectangle that stands for the specified product line. The second level describes possible features, which can be separated into two categories. The first category encloses features such as *Address book* and stands for functionalities, which are part of the platform and can be found inside of any derived product of the mobile phone-SPL. The two remaining features named with *Data transfer* and *Camera* at the second level describe variation points and their representing rectangles are filled out in light grey. The dots on top of each rectangle depict whether a feature is optional (filled out in white) or required/mandatory (filled out in black).

Features can also consist of features and feature groups as shown in the third level. However, feature groups can be selected together at it is the case in the *Address book* feature. Thus, it is possible to have a product of the mobile phone product line that possesses contact data in form of text and a contact photo at the same time (AND feature group). Furthermore, there is a notation to express features that can be selected together. Please note that it is required to select at least one feature (OR feature group). The camera, for example, can have a zoom and also a self-timer. In addition, there are feature groups which cannot be selected at the same time, as it is the case in the feature representing the data transfer functionality (XOR feature group). Furthermore, the *require* arrow from the feature *Contact photo* to the camera feature symbolises that *Contact photo* requires the *Camera* feature, thus, it cannot be selected without the latter.

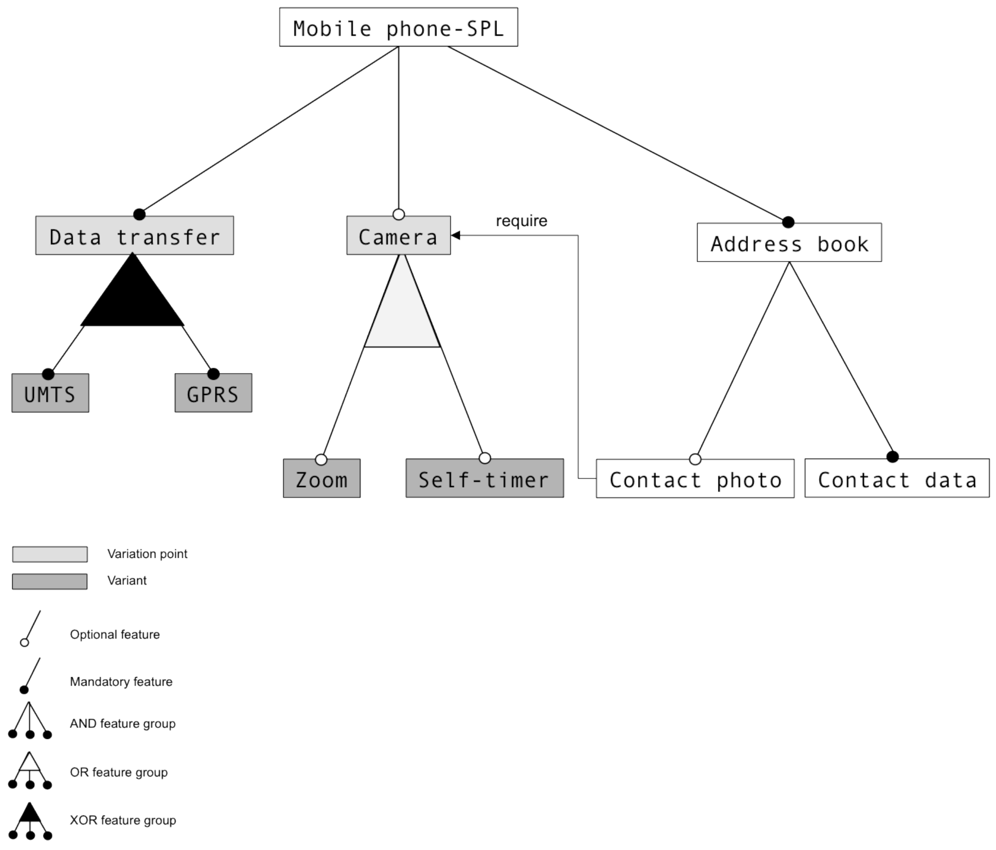


Figure 7: Feature diagram for the mobile phone SPL

The decision modelling approach was developed by the Software Productivity Consortium [29] for industrial use and will be introduced in the following section. The variability model in Table 2 depicts a decision model in a tabular notation adapted from [27] for the mobile phone scenario.

Decision name	Description	Type	Range	Cardinali-ty/Constraint	Visible/Relevant if
Data _transfer	Which data trans-fer standard shall be supported?	Enum	UMTS/GPRS	1:2	-
Camera	Support camera?	Boolean	true/false	-	-
Cam-era_zoom	Support camera zoom?	Boolean	true/false	-	Camera == true
Camera_self-timer	Support camera self-timer?	Boolean	true/false	-	Camera == true
Ad-dress_book_photo	Support contact photo?	Boolean	true/false	-	Camera == true

Table 2: Decision model for the mobile phone SPL.

Obviously, the decision model illustrates only variability aspects of derived products of the mobile phone SPL, however, the feature model defines commonalities as well.

In [30], Bosch et al. identify five level of variability in software product lines such as a *product-line level*, an *architecture level*, a *component level*, a *sub-component level*, and a *code level*.

In a *product line level* variability occurs in terms of the variation of derived products. Additionally, in the *architecture level* or *product level* variability aspects occur in the choice of components and the architecture of a product.

A component is composed of feature sets. Furthermore, variability on the *component level* is concerned with the evolution and the apposition of new implementations in terms of the component interface. On account of the level mentioned previously, the feature sets of a component in the *sub-component level* are selected based on a derived product.

In addition, in the *code level* most variability aspects occur in terms of different products. This is due to the fact that in implementations, respectively in the depending code, particular features of a product are selected. Thus, at this level the difference in detail between products manifests. On account of that, code annotations, so called *ifdef* statements, separate different implementations to identify selected features and their corresponding code. [30]

There are a huge number of tools to define commonalities and variability among products of a product line. Such tools can support stakeholders by configuration and derivation of product variants. Gears [31] and pure::variants [32], are commercial product line tools, which support code annotation in terms of selected features. [33]

In his book [2], Gomaa introduces a modelling approach for software product lines with the UML. The approach is called PLUS (Product Line UML-Based Software Engineering). PLUS provides extensions for existing UML modelling approaches, which are used for the development of single systems. Thus, PLUS extends the UML with concepts and techniques to model commonalities and variability in software product lines. By doing this, Gomaa introduces concepts to express features, interactions between components, and provides extensions such as alternative, mandatory, and optional keywords to express variability, respectively variation points of product line architectures.

Additionally, the PLUS approach provides concepts to model variation points by adding abstract classes. This means that de facto there are classes inheriting from the abstract classes, which differ in their concrete implementation depending on a particular product. Furthermore, Gomaa uses the concept of parameterised classes to design variation points and their depending classes.

Related to SPL and variability modelling of features, this thesis focuses on the variability of design decisions of any kind of UML model element. Variability models in terms of feature models or decision models can express commonalities and variability among products of a product line, but not of different design decisions of UML model elements such as class operations or class attributes. Thus, the definition of such ambiguities requires another concept and cannot be expressed through the variability modelling approaches mentioned.

However, Gomaa introduces a concept of optional, mandatory, and alternative UML elements, by focusing on software product lines in terms of their variability aspects. The PLUS approach represents a basic concept to express optional or mandatory aspects to any kind of UML model element. The Ambiguity Concept can be considered as an extension and can be used for the definition of different design decisions of UML elements.

On account of that, this thesis deals with ambiguities concerning any kind of UML model elements. Furthermore, the next chapter 3 UML and Ambiguities introduces the Ambiguity Concept and shows a concrete approach to add ambiguities to UML model elements.

3 UML and Ambiguities

The definition of different design decisions among UML models requires the usage of an expression language to define decisions in a well-defined and consistent fashion. Due to that, this chapter presents an approach, called the Ambiguity Concept [3], and illustrates its usefulness in terms of adding different design decisions, called ambiguities, to UML models.

As a first step, a running example of an mp3 player is introduced to demonstrate the existence of different design decisions in UML models. Furthermore, the Ambiguity Concept and its basic components are explained. In a third step, a derivation of an ambiguity from different design decisions is presented. Following this, relations and dependencies between ambiguities are discussed. Finally, reasoning over UML models with ambiguities, so called ambiguous reasoning, will be explained.

3.1 Illustration and Running Example

In this subchapter, a running example of an mp3 player is introduced to demonstrate the existence of different design decisions in UML models.

A software designer can choose between many possibilities to model such an mp3 player system. To demonstrate where ambiguities in the modelling process can occur, two variants will be illustrated. The mp3 player scenario is reduced to the main functionalities; operations or attributes are omitted if they were not necessary for this illustration.

The first possible variant of an mp3 player scenario is designed with three components illustrated in Figure 8. The MP3Display stands for a simple display where the user can press buttons to play mp3s from a playlist. The second class MP3Player encapsulates the management of an mp3 playlist. The last class, labelled MP3Playlist, holds the mp3s in a playlist. Additionally, it offers an operation to add mp3s to its playlist. Furthermore, there exists a simple undirected relation between MP3Display and MP3Player. While MP3Display calls operations of MP3Player if a user presses a button to play a song, the MP3Player calls the MP3Display by confirming the command. Hence, the association between MP3Player and MP3Playlist describes that MP3Player holds an attribute (playlist) of the type of MP3Playlist.

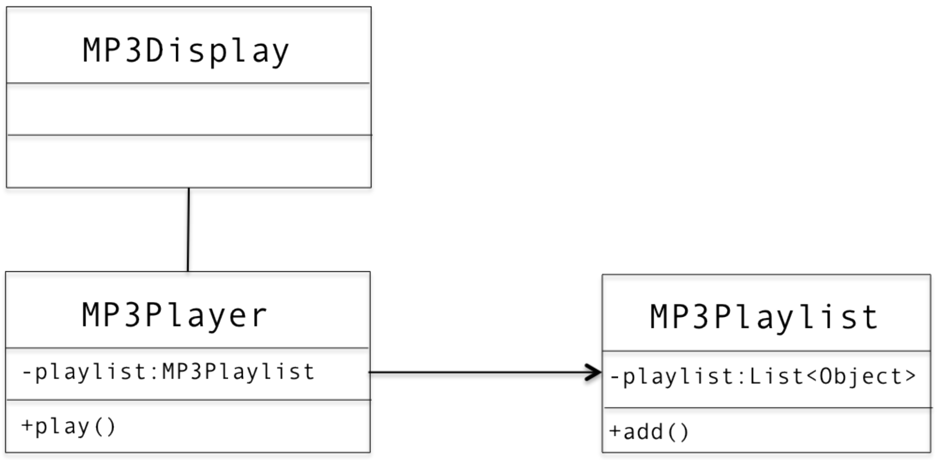


Figure 8: Variant 1: UML class diagram with three components.

Figure 9 shows a Sequence Diagram of the interaction between the three components. If the MP3Display calls the MP3Player to play an mp3, the MP3Playlist is forced to add the selected mp3 to its list via its add operation.

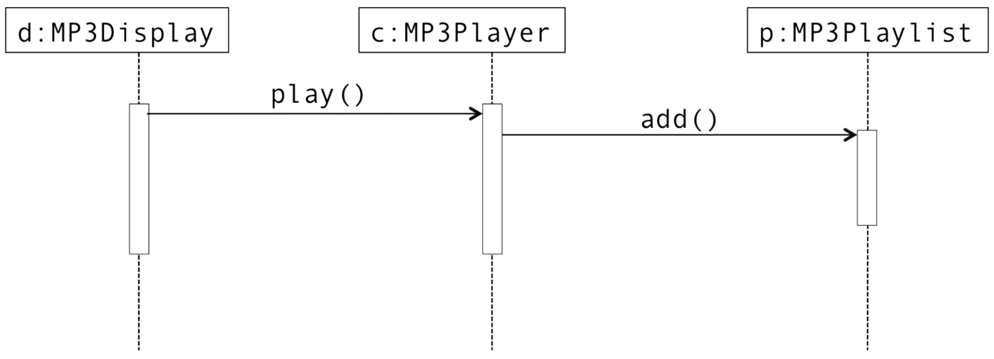


Figure 9: Variant 1:UML sequence diagram with three components.

While this scenario meets all requirements, there might be another possibility for such a scenario. Hence, Figure 10 shows the same scenario without an MP3Playlist class. The functionality of the omitted class is taken over by the MP3Player. Instead of an attribute of the type MP3Playlist, this class has a plain list where the mp3s are stored.

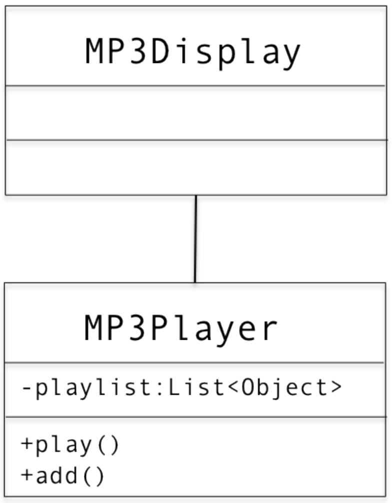


Figure 10: Variant 2: UML class diagram with two components.

Consequently, the interaction between the remaining components changes as illustrated in Figure 11.

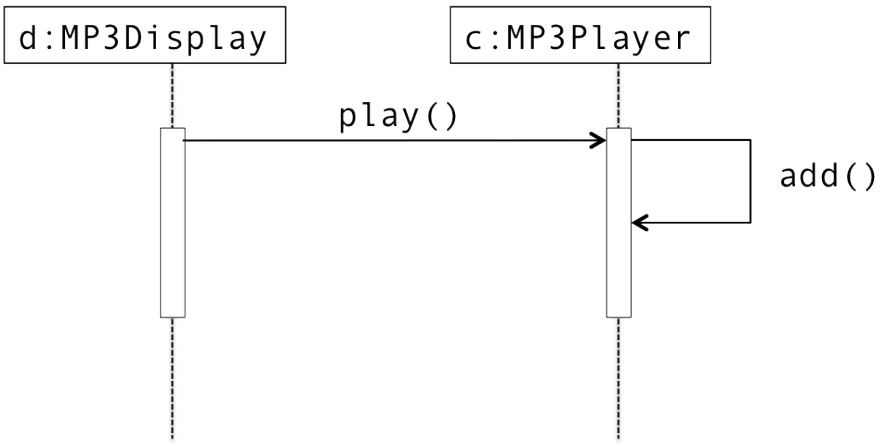


Figure 11: Variant 2: UML sequence diagram with two components.

If a user presses a button to play a song, the MP3Display calls the MP3Player's play operation and its add operation is invoked.

3.2 The Ambiguity Concept

This chapter deals with the Ambiguity Concept and introduces its main components. In [3], Egyed et al. describe an ambiguity as follows:

An ambiguity is a concept that encapsulates an ambiguous design decision for UML elements.

Figure 12 depicts a design snapshot of the Ambiguity Concept and its relations as well as multiplicities among its presented components in UML notation.

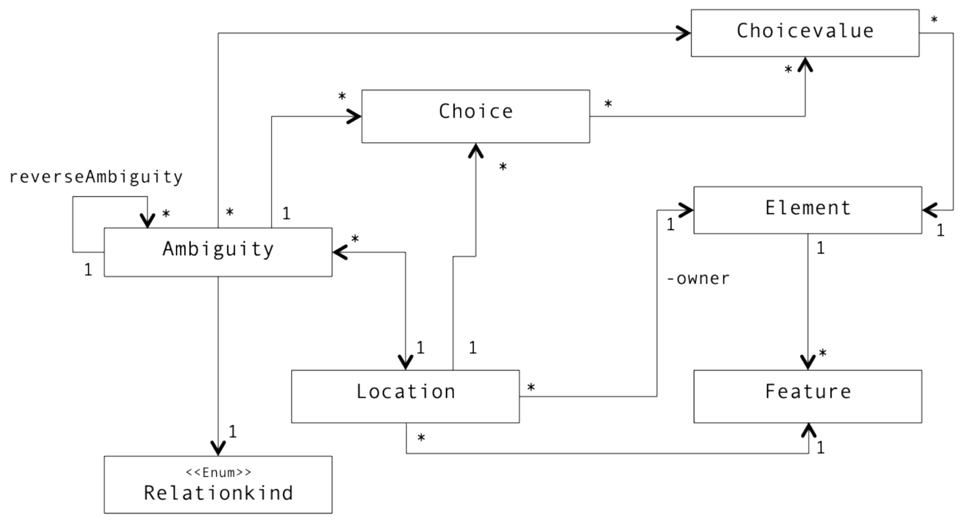


Figure 12: The Ambiguity Concept expressed in UML notation.

Each UML-model element owns certain **features**. Features are structural and behavioural properties, such as associations, class attributes, and operations. Features can consist of complex data types (e.g. self-defined classes) or plain primitive data types (e.g. String, Boolean). However, ambiguities are allowed for any UML element and for any feature.

The **owner** of an ambiguity is simply the UML model element for which the ambiguity was created. Additionally, there exists a component of the type location. A **location** is a triple of the ambiguity owner, the feature, and a list of depending ambiguities. Thus, all ambiguities, which are in a location's list of references must have the same owner and the same feature.

A **choicevalue** is a possible design decision for an ambiguity. A *choicevalue* is a reference of an existing UML model element of the current modelling project, a primitive or a self-defined value that was created for the ambiguity. A *choicevalue* can occur in different *choices* for a location and is a unique element. This means that ambiguities of a location enfolding the same model element as a *choicevalue* reference in fact to the same instance in their *choicevalue* list.

The **relation kind** of an ambiguity describes the relation among *choicevalues* to one another and whether they can be selected together or not. There are three possible values: *xor*, *at_least_one*, and *optional*. In the case of features with a multiplicity greater than one, (e.g. class operations), all three kinds of relation are possible values. In case of a feature with a multiplicity exactly one (e.g. class name), however, there is only *xor* or *optional* allowed. However, *at_least_one* would make no sense, since only one possible value can be selected.

If an ambiguity holds a *xor* relation it means that there is a rule stating that none of the *choicevalues* can be selected together. If this rule is violated, the model becomes inconsistent. Moreover, if an *at_least_one* relation kind is selected, this means that all of the *choicevalues* can be combined together, but at least one of them is required for the model to qualify as consistent. Finally, the *optional* relation kind says that all *choicevalues* can be selected together in every possible manner. Even if none of them is selected the model remains valid.

The algorithm used to create an ambiguity will be described in pseudocode, a simplified programming language that is not defined in a formal way (compared to other programming languages such as Java). Pseudocode facilitates understanding a basic idea of algorithms or a basic approach; less important details are omitted and substituted through non-formal expressions. Thus, it is used for a basic understanding of algorithms and is not appropriate for machinable interpretations. Two slashes symbolise the start of a comment that is not part of the algorithm and has been added for a better understanding of the expression it follows.

The following example in Code 1 depicts the creation of an ambiguity in pseudocode. The function *createAmbiguity* requires four parameters to create an ambiguity. The affected model element (owner) and feature are required to create an ambiguous location. Additionally, the list of *choicevalues* and a relation kind are needed.

```
createAmbiguity(e:UMLElement, f:UMLFeature,
                c:List<Choicevalue>, k:Kind)
begin
   //Calls the basic constructor
   Ambiguity ambiguity = new Ambiguity()
   Location location = new Location(e, f)

   ambiguity.location = location
   ambiguity.relationkind = k
   ambiguity.choicevalues = c
end createAmbiguity
```

Code 1: Creation of an ambiguity in pseudocode.

To provide a concrete example of the creation, the following line shows a call of *createAmbiguity* for the unsolved design decisions of the add operation and its *class* feature introduced in 3.1 Illustration and Running Example.

```
createAmbiguity(add(), class,[MP3Player, MP3Playlist], xor)
```

The result of that call is an ambiguity for the *add* operation, its *class* feature and two possible *choicevalues*, MP3Player and MP3Playlist. As required, the ambiguity describes that the add operation is owned by the MP3Player or the MP3Playlist class.

However, if a *choicevalue* of an ambiguity is represented by an UML model element, in most cases there is a need to describe a reverse relation between the ambiguity owner and the depending *choicevalue*'s feature. Thus, a **reverse ambiguity** is an ambiguity generated automatically for a *choicevalue* of an ambiguity. *Automatically* means that a reverse ambiguity results from the user-defined ambiguity. The feature of a reverse ambiguity is an opposite feature of the original ambiguity feature. However, it is not mandatory that such a reverse relation of features exists.

As mentioned in 2.3.1 Adding Ambiguous Design Decisions, the EMF specifies opposite features for some features, if it is necessary in terms of affections regarding model elements. Thus, in case of the *superclass* feature of a class, there exits e.g. no opposite feature for the related superclass. This is due to the fact that the superclass is not directly affected or depends on the subclass like the subclass is doing on its superclass.

On the contrary, there is the *class* (ownership) feature of an operation and the opposite feature in the EMF specification is *ownedOperation* of a component (class, interface). In this case such an ownership relation exists between both features. This means that an operation is owned by a component.

The relation kind of a reverse ambiguity is always optional and it encapsulates the ambiguity owner of the original ambiguity as the only *choicevalue*. The ambiguity owner of the reverse ambiguity is a *choicevalue* of the original or user-defined ambiguity.

The following pseudocode snipped in Code 1 shows the creation of a reverse ambiguity from a defined ambiguity and a *choicevalue*.

```
createReverseambiguity(a:Ambiguity,
                       c:Choicevalue)
begin
   //Calls the basic constructor
   Ambiguity reverseambiguity = new Ambiguity()
   //Gets the required opposite feature
   Feature oppositefeature = ambiguity.oppositefeature
   //The owner of the given ambiguity
   UMLElement owner = a.owner
   //The choicevalue to create a reverse relation is the owner of //and the
   opposite feature is the feature of the location
   Location reverselocation = new Location(c, oppositefeature)
   reverseambiguity.location = reverselocation
   //Always relation kind optional
   reverseambiguity.relationkind = OPTIONAL
   //the only choicevalue is the owner of the given ambiguity
   reverseambiguity.choicevalues = new List<Choicevalue>(owner)
end createReverseambiguity
```

Code 2: Creation of a reverse ambiguity in pseudocode.

To provide a concrete example of the creation the following line shows a call of *createReverseambiguity* for the above created ambiguity. The opposite feature of an operation's *class* feature is the *ownedOperation* feature of a component. To get that opposite feature, the feature of the ambiguity is required. Thus, the opposite feature can be found retrospectively by a feature's EMF specification.

```
createReverseambiguity(ambiguity, MP3Player)
```

The result of that call is a reverse ambiguity for the MP3Playe*r* class, its *ownedOperation* feature and the single *choicevalue* the add operation.

However, problems in the implementation of the EMF specification with opposite feature definitions exist. This means that there are not defined (null value) opposite features, even if they are required. Thus, it is not possible to generate them introspectively by their class definitions. The opposite feature of a class' *ownedAttribute* feature is not defined as an attribute's *class* feature. Thus, the depending *get* operation simply returns a null value.

To understand the idea of creating ambiguities and reverse ambiguities, some concrete examples are essential. Thus, the following paragraph depicts illustrations and concrete examples of the derivation processes.

Figure 13 illustrates a derivation of an ambiguity with the ambiguity concept. The three rectangles on the left describe how to derive a location from the definition to a concrete instance. Thus, it is mandatory to know, which location in the model is affected. As mentioned before, the triple consists of an UML feature and an UML model element and related ambiguities. In this case is only important to show the derivation of the feature and the model element. Like in the case of the two possible design variants illustrated in Figure 8 and Figure 10 the design decision concerning the add operation's *class* feature can be expressed through an ambiguity. The *class* feature represents an ownership of an operation, respectively to what class an operation depends.

In the first variant illustrated in Figure 8, the add operation is defined in the MP3Playlist. In the second variant shown in Figure 10, there is no MP3Playlist and the MP3Player encapsulates the add operation. While there are two different variants for the same scenario, the same model element and the same feature are affected by an ambiguous design decision. As one can see, the only difference in every variant is the *choicevalue*. MP3Playlist (In the first variant – Figure 8) and MP3Player (in the second variant – Figure 10) are the possible values for the *class* feature and the add operation.

The three rectangles among one another in the middle of Figure 13 describe how to define possible *choicevalues* for an ambiguity. Here two possible *choicevalues* exist: MP3Player and MP3Playlist. Due to the fact that the class feature has a multiplicity of exactly one and that one value has to be chosen[2], the relation kind in this example is *xor* and its derivation is illustrated in the three rectangles on the right.

[2] At least one value is required for an operation and its *class* feature, because an operation cannot be defined without a depending component.

Accordingly, the ambiguity derived in Figure 13 can be expressed in the following line.

```
A1 = (add()+class, xor, {MP3Player, MP3Playlist})
```

The ambiguity can be read as:

The ambiguity A1 for the location add() and its feature class with the relation kind xor has two possible choicevalues MP3Player and MP3Playlist.

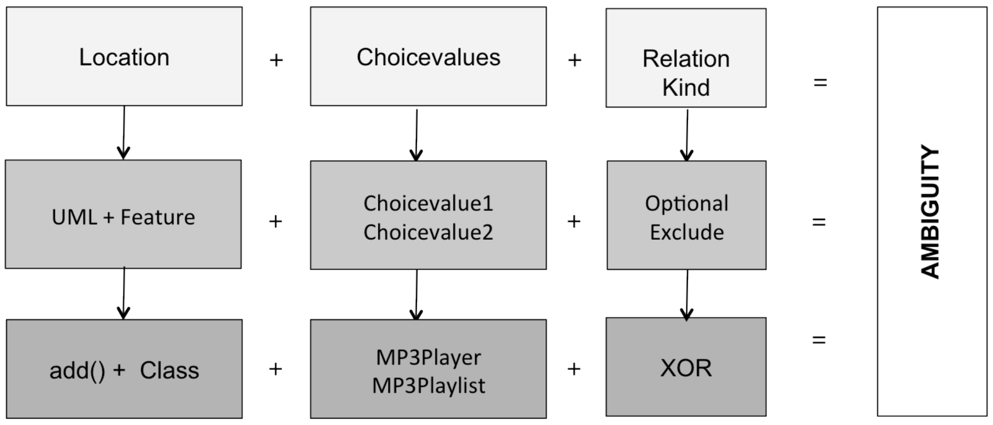


Figure 13: Derivation of an ambiguity

Figure 14 illustrates a derivation of a reverse ambiguity with the Ambiguity Concept for one *choicevalue* of the defined ambiguity in Figure 13. On accord of this, the reverse ambiguity depends on the MP3Player and its feature *ownedOperation*. The only possible value is the add operation and the relation kind is optional.

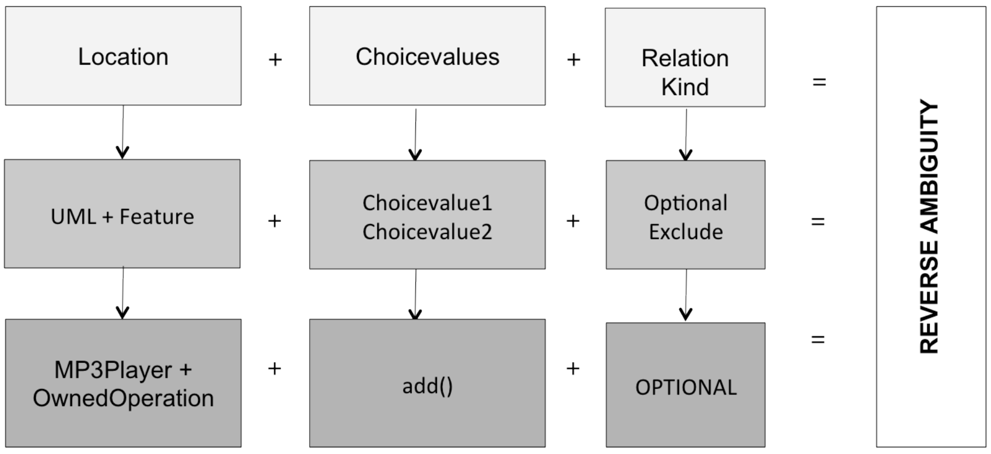


Figure 14: Derivation of a reverse ambiguity

Accordingly, the ambiguity derived in Figure 14 can be expressed as follows.

```
A1.1 = (MP3Player+ownedOperation, opt, {add()})
```

On account of the introduced example of the two different variants of the mp3 player scenario behaviour in Figure 9 and Figure 11, the following ambiguity can be defined.

```
A2 = (add()+receiveEvent, opt, {p, c})
```

This definition expresses that there exists an ambiguity for the location add (message) and its feature *receiveEvent*. This feature describes the target lifeline of a message call. The possible *choicevalues* are p (MP3Player) and c (MP3Playlist). Since only one target lifeline can be selected, the relation kind is *optional*.

There is still one main components of the Ambiguity Concept, called **choice**, left. A *choice* can be considered as a design decision and consists of a set of *choicevalues*. An ambiguity has a set of *choices* generated from its *choicevalues.* A location has *choices* as well and if there is more than one ambiguity for a location, it contains a set of all depending ambiguity *choices* and their combinations. The *choices* of an ambiguity depend on its relation kind.

The *choice* generation for an ambiguity is formulated in pseudocode and is depicted in Code 3. The *choice* generation for locations with more than one ambiguity will be explained in the next subchapter. This section will focus on the *choice* generation for one single ambiguity. Generating *choices* for an ambiguity does not require considering any existing values or so called mandatory values of a location. This will be covered by the generation for a location and will be explained in the next subchapter 3.2.1 Relations between Ambiguities.

However, the algorithm to generate *choices* for an ambiguity is separated into two sections. First, one has to distinguish between features with a multiplicity greater one (*ownedOperation*) and those with a multiplicity exactly one (*class*).

If the multiplicity is greater one, each of the three relation kinds is possible. Furthermore, the result of the *choice* generation depends on the selected ambiguity relation kind. If the relation kind is *optional* or *at_least_one* it is required to determine all combinations of *choicevalues* and to create a *choice* for any combination.

Additionally, if the relation kind is *optional* a *choice* without any *choicevalues* must be created. By doing this, it can be ensured that there is only optionally one of the *choicevalues* taken, but it is not mandatory. Finally, if the relation kind is *xor*, a *choice* for every *choicevalue* must be created separately.

This is also the case if the multiplicity of the feature of an ambiguity is exactly one as described in the second section. Furthermore, if the relation kind is *optional*, a *choice* without any *choicevalues* as explained in the previous section must be created.

```
generateAmbiguitychoices()
begin
       List<Choice> ambiguitychoices = new List<Choice>()
       if isAmbiguityFeatureMultiplicityGreaterOne() then
               if kind == OPTIONAL OR kind == AT_LEAST_ONE then
                   //Create a list of choices, a choice
                   //contains a list of choicevalues.
                   //For each possible combination of all
                   //choicevalues exists a choice
                      createPowerCollection(choicevalues)

                        if kind == OPTIONAL then
                            //Add a choice without choicevalues
```

```
                              addEmptyChoice(ambiguitychoices)
                          end if
                      end if
                      else if kind == XOR then
                         //Add all choicevalues separately in a
                         //choice and add all choices to
                         //the ambiguitychoices list
                         createChoicesForChoicevalues(ambiguitychoices)
                       end if
    //Choices for multiplicity exactly one
    else
       if kind == OPTIONAL then
           //Add a choice without choicevalues
           addEmptyChoice(ambiguitychoices)
       end if
      //Add all choicevalues separately in a choice
      //and add all choices to ambiguitychoices list
       createChoicesForChoicevalues(ambiguitychoices)
    end else
end generateAmbiguitychoices
```

Code 3: Generation of ambiguity choices in pseudocode.

3.2.1 Relations between Ambiguities

There can be more than one ambiguous definition for an UML element and a particular feature. To look upon every possible solution for an UML element and its feature, each ambiguity must be considered separately as well as combined with all others. The combination of an ambiguity with others simply means creating valid combinations of their *choicevalues* regarding their selected relation kinds. In this context *valid* does not mean the set of *choicevalues* will not causes inconsistencies, but that the result is valid in terms of the ambiguities relation kinds.

Accordingly, a *choice* for a location is a solution generated over all owned ambiguities and their *choicevalues*. Thus, a *choice* describes a collection of *choicevalues* for a location. In general, there are no preferences for a certain *choice*. However, the designer has the opportunity to eliminate *choices* by defining constraints. This is part of a reasoner concept and will be discussed later in chapter 3.3 Reasoning over UML Models with Ambiguities.

To understand the basic approach of the algorithm, respectively to generate *choices* for a location, pseudocode is used and is formulated in Code 4.

The original value of an element and its feature, if one exists, have also to be considered here and thus, have to exist in the combinations. The existing original value is called *M* (= mandatory) accordingly to Software Product Lines, where mandatory features cannot be omitted without violating the model.

For a *choice* generation the assumption exists, that there is a mandatory value for the location owner and its feature – otherwise the algorithm would become too complex.

```
generateCombinations()
begin
   //A list for mandatory values of a UML model element and its
   //a feature
   List mandatorylist = new List()
   //All choices of a location
   List choicelist = new List()

   //Add existing values of a feature into a list
   addMandatoryFeatureValue(mandatorylist)
   //If a choicevalue is a mandatory value, it needs to be removed
   //elsewise it cannot be treated as optional
   removeAllChoicevalues(mandatoryList,choiceList)

   //Feature with multiplicity greater one (f.e. ownedOperation)
   if isLocationFeatureMultiplicityGreaterOne() then
      //add a single choice of mandatory values
      addMandatoryChoice(choiceList)
```

```
for i = 0 to allAmbiguities.length
    Ambiguity ambiguity = allAmbiguities(i)
    //Iterate over all ambiguitychoices and merge with choicelist
    for j = 0 to ambiguity.allChoices.length
        Choice ambiguitychoice = ambiguity.allChoices(j)
        //Combine all choicevalues of choices
        for k = 0 to choicelist.length
            Choice locationchoice = choicelist(k)
            //Create a new choice with combinations of
            //choicevalues of ambiguityChoice and
            //locationChoice
            Choice newchoice =
            mergeChoicevalues(locationchoice, ambiguitychoice)
            //Adds the new choice to the choiceslist if there exists
            //no choice with the same choicevalues
           if notExistInChoicelist(newchoice) then
               addToChoices(newchoice)
           end if
        end for
    end for
end for
//Feature with multiplicity exactly one (f.e. class)
for i = 0 to allAmbiguities.length
//Iterate over all ambiguities to add their choices in
//location's choice list
    Ambiguity ambiguity = allAmbiguities(i)
    for j = 0 to ambiguity.allChoices().length
        Choice ambiguitychoice = ambiguity.allChoices(j)
        //Add if there exists no choice with the same
        //choicevalues
        if notExistInChoiceList(ambiguitychoices(j)) then
```

```
                addAmbiguityChoiceToChoiceList()
            end if
        end for
    end for
end if
//Mark choice as invalid if it violates relation kind
findInvalidChoices()
end generateCombinations
```

Code 4 The choice generation in pseudocode.

The generation of all *choices* for a location is separated into two sections. First, the mandatory feature value(s) of the location owner are added together to form a single *choice*. If one of the *choicevalues* of an ambiguity is found in that list, it must be removed from the mandatory list and cannot be considered as a mandatory value any more. This is due to the fact that a *choicevalue* must always be treated as an optional user-defined value. Otherwise it would occur in any *choice* of the depending location. This may result in a location that has only *choices* which are invalid, e.g. if it is based on an ambiguity with relation kind *xor* that says that its *choicevalues* can never be combined without violating the *xor* relation. Due to this, the idea is to remove the *choicevalue* from the mandatory list; this way *choices* with and without that value can exist, leading to valid *choices* in terms of all relation kinds of ambiguities.

The second section is separated into features with a multiplicity greater one (f.e. *ownedOperation* of a UML class) and with exactly one (f.e. ownership between two UML elements). The section for the multiplicity greater one features seems more complicated. In terms of a feature with multiplicity exactly one, each *choice* of all ambiguities is simply added to the location's *choice* list, as long as no *choice* with exact the same *choicevalue* exists.

For a feature with a multiplicity greater one, the main idea is to iterate over all ambiguities for a location and to combine all *choices* of an ambiguity with the existing ones. Accordingly, the iteration starts with a *choice* containing the mandatory values as *choicevalues*. The ambiguity *choicevalues* of each *choice* will be combined with the already generated *choices* of the location. Thus, each *choice* contains the mandatory *choicevalues*. Additionally, there are *choices* with ambiguity *choices* and mandatory *choicevalues*. All *choices* of an ambiguity will be combined with all other ambiguities and their *choices* by merging both *choices*' *choicevalues* together to create a new *choice*.

The last part of a *choice* generation is to filter all generated *choices* and to invalidate them if one of them violates a relation kind of any ambiguity.

The following simplified example shows how to generate *choices* of a location with a single ambiguity and among different ambiguities to get valid *choices* in terms of features with multiplicity greater one. Features with a multiplicity exactly one are not considered any more, because their *choices* always consist of only one *choicevalue*.

All important information of ambiguities, which are owned by a location to conduct a *choice* generation, is formulated by the following expression:

```
L → {(xor, {y,z}), (opt, {a,b})}
```

That expression can be read as:

For location L there exists an ambiguity with a relation kind xor, choicevalues y and z; there exists a second ambiguity with relation kind optional, choicevalues a and b.

The ambiguity for location MP3Player+OwnedOperation, with the relation kind optional, and the single *choicevalue* add(), taken from Figure 14 expressed in the same way reads as follows:

```
MP3Player+ownedOperation → {(opt, {add()})}
```

In the given example the mandatory value (M) for the location MP3Player+OwnedOperation is the operation play. This operation exists in both variants of the defined mp3 player scenario illustrated in Figure 8 and Figure 10. Figure 15 depicts the two *choices* symbolised by rectangles, which are derived from the mandatory value and the defined ambiguity. Since the relation kind of the ambiguity has the value *optional*, the first *choice* holds only the mandatory value *play()*. The second *choice* has two *choicevalues*: *play()* and *add()*. The two *choices* are valid in terms of the relation kind of the defined ambiguity, because in both choices the *add* operation exists or does not exist. This is the definition of the optional relation kind value in a nutshell. Additionally, there always exists a mandatory value, in this case the play operation.

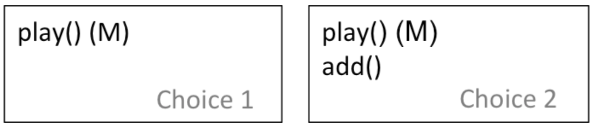


Figure 15: Choices for a single ambiguity.

While the *choice* generation of a location with a single ambiguity seems very easy, generating *choices* of more than one ambiguity for a location is more complex. To exemplify a more complex aspect it is mandatory to define another ambiguity for the location *MP3Player+OwnedOperation.* The following ambiguity describes the design decision of another operation in the MP3Player class.

```
MP3Player+ownedOperation → {(xor, {pause(), stop()})}
```

The design decision is ambiguous because of the mutually exclusive operations pause and stop. The pause operation offers the functionality to pause and continue playing a current mp3. However, the stop operation just stops a current mp3 song; if the play operation is called, the mp3 will start at the beginning. Of course it would be more user-friendly to offer both opportunities, but for this simplified mp3 player scenario, one opportunity suffices.

The ambiguities can be expressed as follows:

```
MP3Player+ownedOperation →
{(opt, {add()}),(xor, {pause(), stop()})}
```

Figure 16 shows the *choices* of both combined ambiguities.

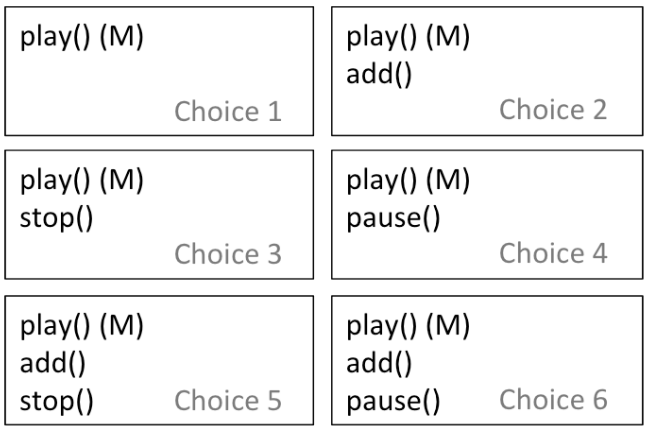


Figure 16: Choices for two combined ambiguities.

Based on the two ambiguities and their relation kinds, there are six *choices*. The first and the second *choice* also exist in the *choices* of the single ambiguity. In the next step the question whether they are consistent in terms of the second ambiguity will be discussed. First, they are used to generate the next *choices* for the second ambiguity. *Choice 1* is now combined with the *choicevalues* of the second ambiguity *stop()* and *pause()*. Since both operations cannot exist in the same *choice* (because of the relation kind *xor*) they are combined separately with the mandatory value *play()* (*choice 3* and *choice 4*). Analogous to that, *choice 5* and *choice 6* are generated by combining *choice 2* with all *choicevalues* of the second ambiguity *stop()* and *pause()*.

As mentioned before, it is important to know which *choice* is valid for each defined ambiguity and its relation kind. In the case at hand, the second ambiguity has a relation kind value *xor*; and if a *choice* without exactly one of the ambiguity's *choicevalue* existed, the ambiguity relation kind would be violated. In c*hoice 1* and *choice 2* there is neither a stop operation nor a pause operation. This may not cause inconsistencies in the UML model, but it will cause them in the designer's ambiguities definitions. Thus, both *choices* invalidate the second ambiguity and can be omitted in any further validation or reasoning steps.

Another possibility of generating invalid *choices* is the existence of ambiguities where *choicvalues* would be mutually exclusive, but end up in one *choice* due to the combination of two ambiguities. This can happen if equal *choicevalues* exist in more than one ambiguity for the same location while featuring a relation kind value *xor*. To illustrate this problem the following two ambiguities are defined:

```
MP3Player+ownedOperation → {(opt, {pause()})}

MP3Player+ownedOperation → {(xor, {pause(), stop()})}
```

The first ambiguity for the location *MP3Player+OwnedOperation* has an optional pause operation, the second one features two mutually exclusive operations called pause and stop. In addition, Figure 17 depicts the combined *choices* of both ambiguities' *choicevalues*. The first and the second *choice* of the first ambiguity are combined with the *choicevalues* of the second ambiguity, hence, stop and pause.

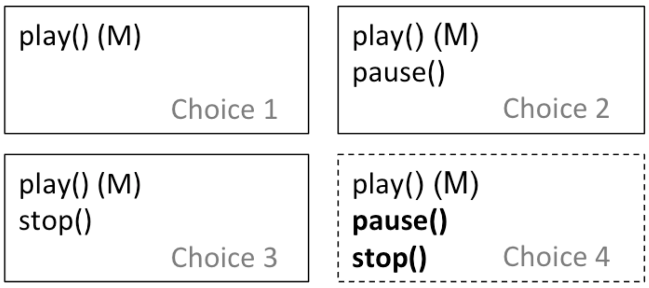


Figure 17: Example of invalid choices for two combined ambiguities.

Due to the fact, that there exists a pause operation in both ambiguities, it also occurs in the second *choice*. Combining the two ambiguities leads to *choice 4*. In this *choice*, pause and stop are selected together in a single *choice*, thus invalidating the relation kind value *xor* defined in the second ambiguity.

3.3 Reasoning over UML Models with Ambiguities

The previous section dealt with the Ambiguity Concept and introduced its main components. Furthermore, it described how Egyed et al. developed an approach to reason over design models with ambiguities, *choices*, and consistency rules, called *Ambiguous Reasoning* (AR) [3]. This chapter deals with the algorithm of AR and provides examples to illustrate how the reasoning process is conducted.

It is essential to assume that there exists a model repository where the defined model and its components are stored. Furthermore, all ambiguities have to be defined with the Ambiguity Concept and are complete. *Complete* means they depend on a location have a set of or at least one *choicevalue(s)* and a relation kind. In Addition, all *choices* are generated as explained in 3.2 The Ambiguity Concept.

The particular algorithm and its basic concept taken from Egyed et al. [3] will be explained in the context of CSPs. In terms of a CSP, ambiguities can be considered as variables and different design decisions, so called *choices*, as values of a domain. Further, in a CSP a constraint is a condition that returns *true* or *false* for a given set of variables.

However, in a consistency rule for an UML model there are no exact definitions of possible values that must be fulfilled: it is a more generic approach. That means that a rule for an UML model de facto *provides navigation instructions* to identify affected elements and properties. [3] Thus, an ambiguity and its *choices* are not defined as a set of possible values, but through the validation; *choices* can be encountered and can be used to navigate through a model as well.

An ambiguity and its *choices* can be considered as unique elements. Egyed et al. introduce the type *pairings*. A *pairings* represents one or more sets of an ambiguity and a *choice* pair.

```
Parings = {(Ambiguity, Choice1), (Ambiguity, Choice2)}
```

In the case of the example illustrated in Figure 13 and its derived ambiguity, there exists the following set of *parings*:

```
A1 = (add()+class, xor, {MP3Player, MP3Playlist})
 ParingsA1 = {(A1, MP3Player),(A1, MP3Playlist)}
```

There are two *pairings* for the ambiguity called A1. Furthermore, they can be used to define so called *determinations* (Egyed et al. called it *Assignments*) for the validation process. A *determination* is a selection of a particular *pairing* taking place if an ambiguous location is found during an evaluation. Thus, a *determination* is a function that takes an ambiguity and returns a particular *choice*. It can be expressed as follows:

```
Determination = Ambiguity → Choice
```

A concrete *determination* for ambiguity A1 is defined as:

```
Determination1 = A1 → MP3Player

Determination2 = A1 → MP3Playlist
```

Additionally, a constraint can be evaluated to a Boolean condition, e.g. in a CSP. As mentioned before, in a CSP a constraint is a condition that returns *true* or *false* over a given set of variables. Furthermore, a *determination* can be used to evaluate a constraint with the *determination's* particular *choice* to *true* or *false*. Thus, a constraint is evaluated by taking the *choice* as input and as current value for the current evaluated location.

Concerning the reasoning process, a particular reasoner takes a model element from a model repository and a consistency rule as input and performs reasoning steps. Hence, the AR mechanism applies the model element from the repository if there no ambiguity or reverse ambiguity for the current model element and feature (location) exists. If there such an ambiguous location exists, the generated *choices* are used instead of repository values and the evaluation is performed with any *choice*.

Due to the fact that elements affect one another a *choice* and a depending *choicevalue* that is ambiguous as well can exist at the same time. The reasoner works iterative, meaning, that it stops if an ambiguous *choicevalue* and feature is found that affects the current location. In this case it performs the same evaluation with all depending *choices* for the new ambiguous location (*choicevalue*) and so on.

If the evaluation of a *choicevalue* returns a negative result, the whole *choice* and all other *choices*, which enfold the same *choicevalue* are marked as invalid. Furthermore, the reasoning process must be conducted again to invalidate all those *choices* enfolding the invalidated *choicevalues*. Since *choicevalues* can occur in many *choices*, it is necessary to keep track of all occurrences, so that the remaining *choices* can be marked as invalid as well.

The following pseudocode snipped in Code 5 depicts the AR algorithm in a simplified way.

```
validate(Constraint c, Determination d)
begin
   //Evaluate constraint with determination
   isSatisfied = evaluate(c, d)
   if isSatisfied then
       //Add d to all positive determinations for c
       satisfiedDeterminations(c).add(d)
   end if
   //Ambiguous location found
   if ambiguityFound() then
       //Take choices and validate them
       for i = 0 to choices.length
           Choice choice = choices(i)
           if isChoiceValid() then
              validate(c, new Determination(choice))
              if d.hasInvalidValues()
                 choice.invalidate()
                 //Simplified: validate all positive determinations
                 //for c again, because it may be the case that
                 //evaluated choices are now invalid, respectively
                 //contains of invalid choicevalues.
                 validate(c, satisfiedDeterminations(c))
              end if
           end if
       end for
   end if
end validate


evaluate(Constraint c, Determination d)
begin
   //Values = choicevalues
   hasInvalidChoicevalue = evaluateChoicevalues(c,d)
```

```
    if hasInvalidChoicevalue then
        //Invalidate choicevalue to eliminate a choice
        invalidate(d.invalidChoicevalues)
    end if
end evaluate
```

Code 5: Ambiguous Reasoning in pseudocode.

However, to understand that evaluation process an example is essential. The first rule of Table 1 affects the introduced sequence diagrams in Figure 9 and Figure 11, respectively all message calls between lifelines. Accordingly, that rule defines that the name of a message call must be an operation in the receiver's class definition.

The first call named play between the lifelines d and c must be evaluated in terms of the first rule. Since the evaluation of the second call named *add()* between c and p is more complex, the detailed explanations of the first call *play()* will be omitted.

As a first step one needs to define all necessary ambiguities for the two variants of the mp3 player scenario. The ambiguity expressing the variable target lifeline of the two sequence diagrams in Figure 9 and Figure 11 can be expressed as follows:

```
A2 = (add()+receiveEvent, opt, {c, p})
```

Additionally, there are two different class definitions of the MP3Player and MP3Playlist in Figure 8 and Figure 10. The ambiguity expressing the different class values for the add operation is defined as:

```
A1 = (add()+class, xor, {MP3Player, MP3Playlist})
```

The depending reverse ambiguities are:

```
A1.1 = (MP3Player+ownedOperation, opt, {add()})
A1.2 = (MP3Playlist+ownedOperation, opt, {add()})
```

The evaluation process is conducted based on the four ambiguities. Figure 18 illustrates the reasoning process.

There are two generated *choices* with single *choicevalues* c and p. For each *choice* a *determination* is derived on the base of which the first rule is evaluated. To evaluate this rule its definition states that in the operations of c and p an operation named with the same name of the investigated message call must exist. The message call name is add.

The reasoner evaluates the rule for the first *choice* c of type MP3Player. Thus, it searches in all values for the feature *ownedOperation* of the class definition for the MP3Player. Since the feature is ambiguous (due to the reverse ambiguity `A.1.1`) the same evaluation process is conducted for that location.

With respect to the mandatory value *play()* there are two *choices* of `A.1.1`. The first enfolds only the play operation, the second has two *choicevalues*, namely *play()* and *add()*.

To get back to the reasoning process, the first *choice* is used to evaluate the first rule. The created *determination* with that *choice* can be considered as invalid. This is due to a missing add operation in that *choice*. Accordingly, the second *choice* (*play()*, *add()*) is applied. Here a *choicevalue*, respectively an operation named with add, exist and the *determination* returns a valid result. Thus, the first *choice* with *choicevalue* c is valid for the *receiveEvent* of the message *add()*.

The reasoner do returns to the point where the next *choice* p must be evaluated. The reasoning process for the second *choice* p is realised based on the first *choice* p. Due to the defined ambiguity `A.1.2` an ambiguous location is encountered by searching for an add operation in the values of the *ownedOperations* feature of p.

Since there is no mandatory value the first resulting *choice* of `A.1.2` is an empty *choice*. This means that it contains no *choicevalue* in it. This is possible due to the fact, that the add operation in `A.1.2` is defined as optional. The second resulting *choice* enfolds the add operation.

Due to an existing add operation in the second *choice*, the reasoner evaluates the created *determination* to true. However, the first *choice* without an add operation is evaluated to false.

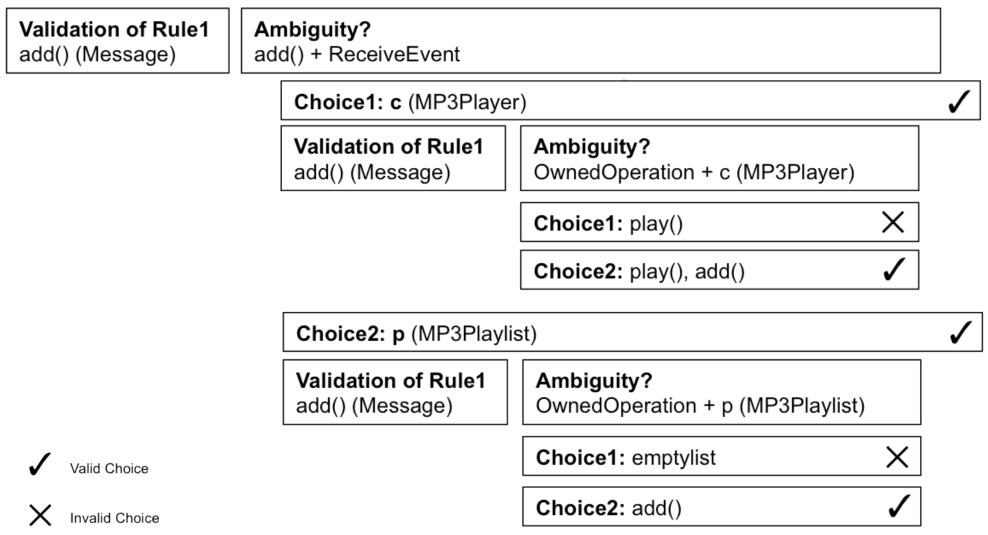


Figure 18: Reasoning process of the first consistency rule.

In conclusion, the model with respect to all defined ambiguities can be considered as consistent related to the first consistency rule. However, the consistency is only guaranteed with adequate *choices*. Thus, AR can be used to show what *choice* is a valid selection in case of design decisions or defined ambiguities affecting one another. Due to that, in practice the designer can be supported in terms of the selection of a particular decision and can reject incompatible decisions.

4 Implementation

This chapter deals with the implementation of the *Ambiguitymanager* tool, a plugin for the Rational Software Architect. The *Ambiguitymanager* realises the Ambiguity Concept and provides ambiguity management. In addition, the *Ambiguitymanager* is linked to an existing model analyser tool. Together they can be used to conduct consistency checks, i.e. checking if the model remains valid with different combinations of design decisions.

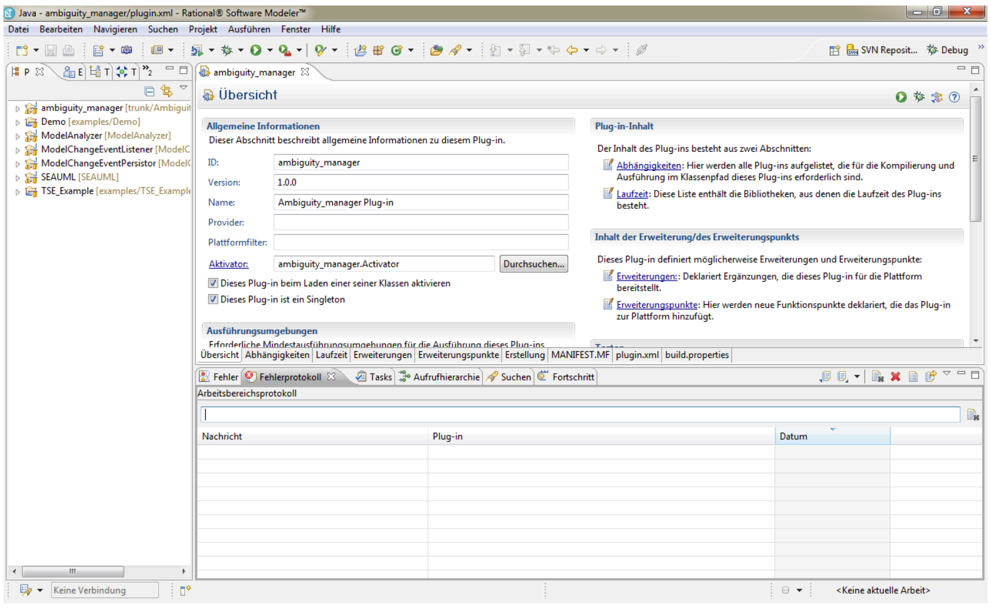
First, an overview about main functionalities of the Rational Software Architect will be provided. In a second step the *Ambiguitymanager* and the core requirements of that plugin will be pointed out. In the following chapter, reasoning results so called *determinations* will be explained. In addition, useful information for a designer that can be derived from that output will be mentioned. Finally, limitations of the implementation and the connection to an existing model analyser will be described.

4.1 Rational Software Architect

In this subchapter the IBM Rational Software Architect (RSA) [16], a modelling tool for software systems, will be introduced and an overview about its architecture will be provided.

The RSA is a modelling tool for software systems and property of the IBM Rational Software [17] division. With the RSA a software designer can specify architectural and behavioural properties of a system with the UML. A software designer has the ability to choose from a huge number of representations of development artifacts offered by the RSA. [3]

The RSA is an Eclipse-based modelling tool and contains perspectives such as e.g. a *Java*, a *Debug*, and a *Repository* perspective also a *Modelling* perspective in addition to the standard Eclipse [34]. The screenshot depicted in Figure 19 shows the *Java* perspective of the RSA.


Java - ambiguity_manager/plugin.xml - Rational® Software Modeler™
Datei Bearbeiten Navigieren Suchen Projekt Ausführen Fenster Hilfe
SVN Reposit... Debug
P E T T
ambiguity_manager [trunk/Ambiguit
Demo [examples/Demo]
ModelAnalyzer [ModelAnalyzer]
ModelChangeEventListener [ModelC
ModelChangeEventPersistor [ModelC
SEAUML [SEAUML]
TSE_Example [examples/TSE_Example
ambiguity_manager
Übersicht
Durchsuchen...
Übersicht Abhängigkeiten Laufzeit Erweiterungen Erweiterungspunkte Erstellung MANIFEST.MF plugin.xml build.properties
Fehler Fehlerprotokoll Tasks Aufrufhierarchie Suchen Fortschritt
Arbeitsbereichsprotokoll
Nachricht Plug-in Datum
Keine Verbindung
<Keine aktuelle Arbeit>

Allgemeine Informationen

Dieser Abschnitt beschreibt allgemeine Informationen zu diesem Plug-in.

ID: ambiguity_manager
Version: 1.0.0
Name: Ambiguity_manager Plug-in
Provider:
Plattformfilter:
Aktivator: ambiguity_manager.Activator

[✓] Dieses Plug-in beim Laden einer seiner Klassen aktivieren
[✓] Dieses Plug-in ist ein Singleton

Ausführungsumgebungen

Erforderliche Mindestausführungsumgebungen für die Ausführung dieses Plug-ins

Plug-in-Inhalt

Der Inhalt des Plug-ins besteht aus zwei Abschnitten:

Abhängigkeiten: Hier werden alle Plug-ins aufgelistet, die für die Kompilierung und Ausführung im Klassenpfad dieses Plug-ins erforderlich sind.

Laufzeit: Diese Liste enthält die Bibliotheken, aus denen die Laufzeit des Plug-ins besteht.

Inhalt der Erweiterung/des Erweiterungspunkts

Dieses Plug-in definiert möglicherweise Erweiterungen und Erweiterungspunkte:

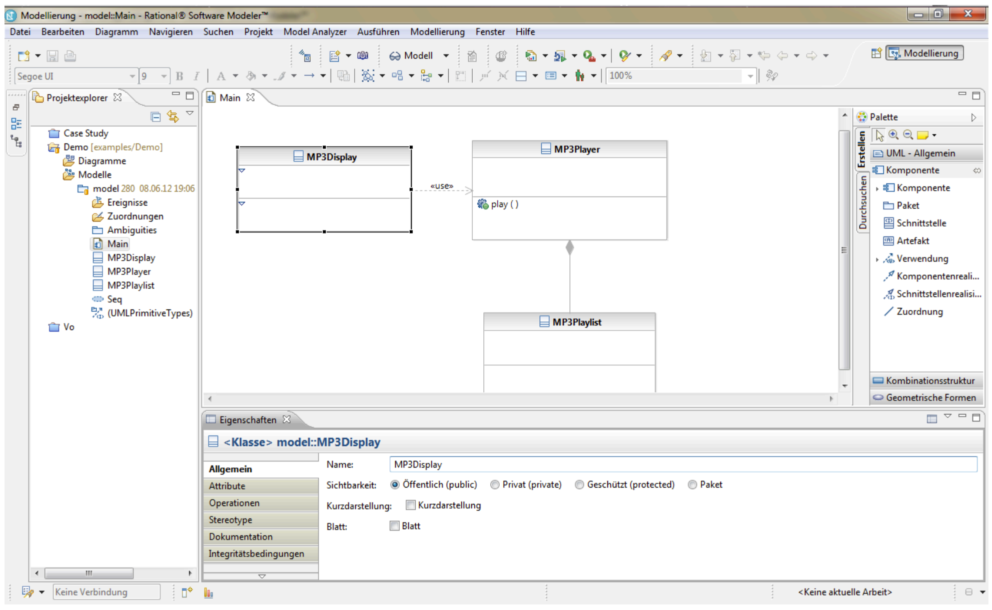
Erweiterungen:: Deklariert Ergänzungen, die dieses Plug-in für die Plattform bereitstellt.

Erweiterungspunkte: Hier werden neue Funktionspunkte deklariert, die das Plug-in zur Plattform hinzufügt.

Figure 19: The Java perspective of the RSA.

As visible above, the architecture of the user interface looks similar to a plain Eclipse IDE with a *Project Explorer* on the left and different views at the bottom such as a *Call Hierarchy* view and a *Tasks* view.

As mentioned before, the RSA also offers a *Modelling* perspective depicted in Figure 20. The *Project Explorer* on the left shows current modelling projects and all UML elements in a tree structure. An editor for graphical representations of UML model elements and their properties is situated in the middle. Several other views are placed at the bottom; one example is the *Property* view enabling the user to manage and browse through model elements' properties.



<Klasse> model::MP3Display

Name: MP3Display

Sichtbarkeit: [✓ Öffentlich (public)] Privat (private) Geschützt (protected) Paket

Kurzdarstellung: ☐ Kurzdarstellung

Blatt: ☐ Blatt

Figure 20: The Modelling perspective of the RSA.

On the right there is a pallet of a range of UML model elements which can be inserted into a project via the drag and drop mechanism.

4.2 The Ambiguitymanager Plugin

In chapter 3 UML and Ambiguities the Ambiguity Concept was introduced and a running example providing a detailed description of the process of adding ambiguities to UML design models was presented. As mention in chapter 2.3 Background, Egyed et al. [3] first came up with the idea of adding unsolved design decisions to UML models.

While they developed a plugin for the RSA to manipulate UML elements such as classes, lifelines or class operations in terms of their UML features, there remain some weaknesses in terms of the realisation and a missing graphical user interface.

One weakness of the plugin is that all different design decisions, so called ambiguities, must be added manually and directly into the program code. There is no graphical user interface facilitating the adding process. Additionally, the output of the reasoning process is not presented in a user-friendly manner and all reverse dependencies, so called reverse ambiguities for model elements, are not generated automatically.

Based on these issues, the current chapter deals with the development of a plugin for the RSA called *Ambiguitymanager* that will be connected to an existing model analyser plugin to solve the weaknesses of Egyed et al. [3].

4.2.1 Requirements

In this subchapter, the five main requirements will be presented and their implemented solution will be explained.

The five core requirements are:

1. Graphical user interface: Design and implementation of a graphical user interface
2. Persistence: Persistence of ambiguities into a modelling project
3. Reverse ambiguities generation: Automatic generation of the reverse ambiguities
4. *Choice* generation: Automatic generation of *choices* for a location
5. Model analyser connection: Connect plugin to existing model analyser

The first requirement describes the necessity of a graphical user interface to add ambiguities to UML models. The second requirement defines that ambiguities must be saved permanently into the current modelling project, thus enabling users to access them later on. The third and fourth requirements deal with the solution about how to generate reverse ambiguities and *choices* automatically, as discussed in chapter 3.2 The Ambiguity Concept. The final requirement handles the connection between the *Ambiguitiymanager* tool and an existing model analyser to perform reasoning steps as introduced in 3.3 Reasoning over UML Models with Ambiguities.

4.2.1.1 Graphical User Interface

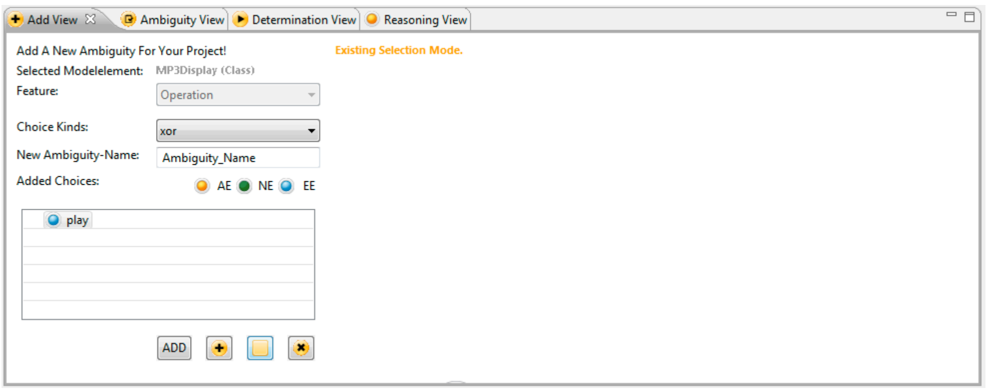
The first requirement describes the necessity for a graphical user interface to manage ambiguities of UML models. Designing and implementing a graphical user interface means developing an intuitive and user-friendly interface for the RSA, allowing a user to easily manage ambiguities for UML elements. That means, in detail, to provide a visualisation of the defined ambiguities, the generated reverse ambiguities as well as *choices* to the designer, to let him track reasoning steps, and to present him an overview about the reasoning output, the so called *determinations*.

The *Ambiguitiymanager* plugin consists of four views of the type *ViewPart* [35].

The AddView

With the first view, as depicted in Figure 21, the designer is enabled to add ambiguities to an UML model element. Additionally, this view is also used to edit ambiguities.

The designer can chose the UML element simply by selecting it in any editor of the RSA.



Add A New Ambiguity For Your Project!
Existing Selection Mode.
Selected Modelelement: MP3Display (Class)
Feature: Operation
Choice Kinds: xor
New Ambiguity-Name: Ambiguity_Name
Added Choices:
play

Figure 21: The AddView to add ambiguities to UML model elements.

The possible features of the selected element a user can create an ambiguity for are available in the first combobox. They were defined via an XML file for each UML model element. The decision for a XML driven feature definition was made due to problems in the EMF specification. This means that there are problems with undefined (null value) features for the reverse ambiguities. Thus, they need to be defined manually – it is impossible to generate them introspectively by their class definitions. On account of that, the opposite feature of a class and its *ownedAttribute* feature is not defined as the attribute's *class* feature, thus, the depending *get* operation simply returns a null value.

The feature definition and the persistence of ambiguities are realised with the same XML file. The persistence of ambiguities will be discussed later in this subchapter. The following simplified code snippet Code 6 shows such a definition for a UML class and its features *isAbstract*, *ownedOperation*, and *ownedBehavoir*.

```
<Class literal="CLASS">
    <Feature label="is Abstract" name="isAbstract" />

    <Feature label="Operation" name="ownedOperation"
     opposite="class" />

    <Feature label="Behavior: state machine" name="ownedBehavior"
     opposite="owner" value="STATE_MACHINE"
     diagramKind="STATECHART_LITERAL" />
</Class>
```

Code 6: XML feature definition.

The first entry after the *xml tag* describes the UML element type for which the following *feature tags* are defined. The *literal* [36] *tag* has the value "CLASS" and represents UML classes. The next tags are used for the definition of UML features of model elements. The first *feature tag* represents the *isAbstract* feature of a class that is usually visualised through the UML keyword *{abstract}* over a class name in a class rectangle and thus describes if the class is abstract.

The first xml attribute of the *feature tag* is called *label* and is used only for the display in the feature combobox. The *name attribute* represents an original EMF feature reference name. Since the *isAbstract* feature has always a primitive value (Boolean) and no reverse ambiguities for *choicevalues* must be created, there is no need to make any statements about an opposite feature.

The second *feature tag* represents the *ownedOperation* feature of a class, thus the *name attribute* holds that EMF feature name. Furthermore, the value of the *attribute label* is *operation*. The feature *ownedOperation* describes operations of a class. An operation has a complex class type for which a definition for its reverse ambiguities and thus its opposite feature must exist. In this case the opposite feature of the *ownedOperation* feature is the operation's *class* feature (ownership of an operation).

The last *feature tag* defines the *ownedBehavoir* feature of UML classes. This feature represents behavioural properties of a class and is visualised through a state machine. The *feature tags label*, *name*, and *opposite attributes* are defined in a similar manner as in the previous feature. The *value attribute* describes a type for a new added *choicevalue* and is represented by a reference literal. The *diagramKind attribute* is required if the *choicevalue* must be contained in a certain diagram as it is here the case.

Continuing with the description of the usage and functionalities of the *AddView*, the designer can select an ambiguity relation kind by choosing the appropriate one from the second combobox. He can also type ambiguity's name and add new *choicevalues* or select one of the existing matching UML model elements from the whole model project. *Matching* means here that the selected UML element and its type must be compatible with the selected feature and its required element type.

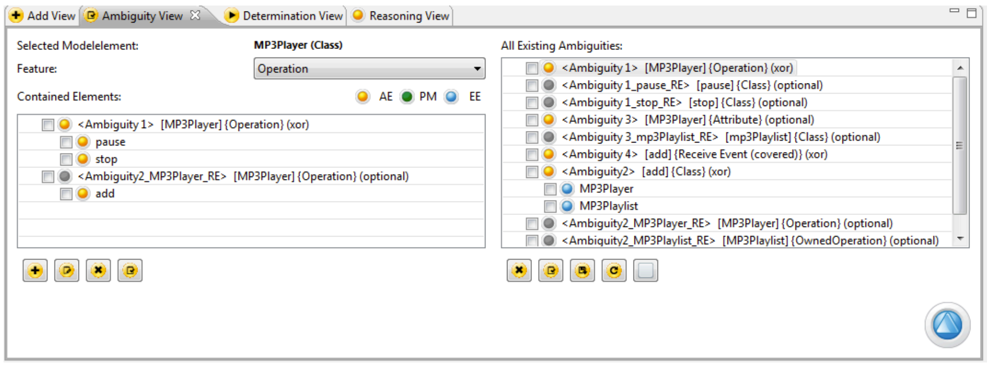
As a example, let us assume the *choicevalue play()* of the following ambiguity existed in another class and the designer would like to select it as a *choicevalue* for the MP3Player and its feature *ownedOperation*.

```
MP3Player+OwnedOperation → {(opt, {play()})}
```

The required element type of this feature is an UML *operation* and thus, the designer should only select UML model elements from the same type. This is ensured by only adding elements if their type is compatible.

The AmbiguityView

The second view implemented for the *Ambiguitymanager* is the *AmbiguityView* shown in Figure 22. The *AmbiguityView* offers a designer a detailed overview tree of all defined ambiguities and a separate tree of ambiguities for each model element and feature. The latter updates its input by changing the selection of a model element in any editor of the RSA. Additionally, functionalities exist to delete ambiguities, edit a selected ambiguity (the *AddView* is called with the selected ambiguity), and resolve a selected ambiguity. The latter functionality realises the transformation of an ambiguity definition and the selected *choicevalues* into the depending location.



Selected Modelelement: MP3Player (Class)
Feature: Operation
Contained Elements: AE PM EE

```
<Ambiguity 1>  [MP3Player] {Operation} (xor)
    pause
    stop
<Ambiguity2_MP3Player_RE>  [MP3Player] {Operation} (optional)
    add
```

All Existing Ambiguities:

```
<Ambiguity 1>  [MP3Player] {Operation} (xor)
<Ambiguity 1_pause_RE>  [pause] {Class} (optional)
<Ambiguity 1_stop_RE>  [stop] {Class} (optional)
<Ambiguity 3>  [MP3Player] {Attribute} (optional)
<Ambiguity 3_mp3Playlist_RE>  [mp3Playlist] {Class} (optional)
<Ambiguity 4>  [add] {Receive Event (covered)} (xor)
<Ambiguity2>  [add] {Class} (xor)
    MP3Player
    MP3Playlist
<Ambiguity2_MP3Player_RE>  [MP3Player] {Operation} (optional)
<Ambiguity2_MP3Playlist_RE>  [MP3Playlist] {OwnedOperation} (optional)
```

Figure 22: The AmbiguityView to manage defined ambiguities.

In addition, the designer is offered two possibilities how the *Ambiguitymanager* should behave if a model changes. An example of a model change is the alteration of an element's name. Since there might be *choicevalues*/model elements selected for an ambiguity and visualised with its name, it is necessary to update them and display them again.

The first possibility is to react immediately and automatically on model changes and to update all ambiguities by loading all persistent ambiguities from a project. This is realised through a *listener* that reacts on all model changes.

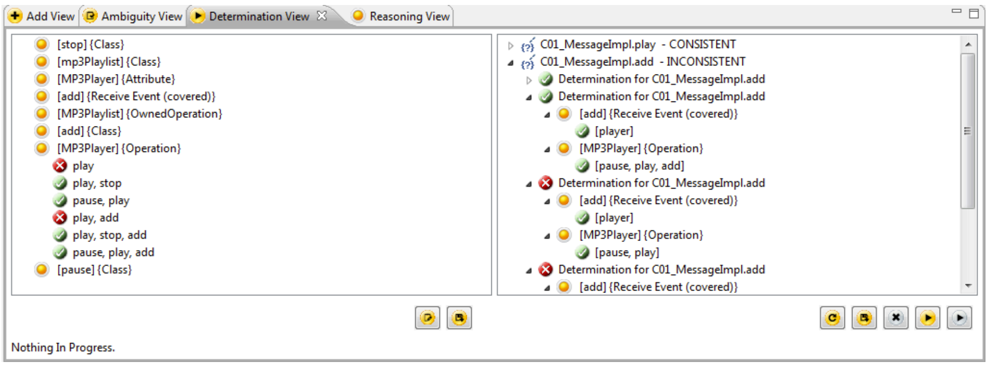
The second possibility is to load all ambiguities manually by clicking an update button. These two possibilities are realised with a button to enable and disable the automatic update function. In projects with a huge number of model elements and a huge number of defined ambiguities it is recommended to update the ambiguities manually. This is due to the fact that the RSA notifies its listeners even in case of minor changes e.g. when a model element changes its position in a class diagram. Since a model changes very often during the modelling phase, updating it even if an update would not be necessary is more expensive in terms of resources.

Finally, there is a functionality to save all defined ambiguities into a plain formatted txt-file. However, this is only important for documentation aspects, since ambiguities are persisted directly into a project folder as well

The DeterminationView

The *DeterminationView* is the third view developed and displays generated *choices* sorted by location and reasoning output, respectively the *determinations* for a rule (constraint) of the reasoning process. This view is depicted in Figure 23.

Additionally, it is possible to save *determinations* and generated *choices* into a plain txt-file. The reasoning process, also known as validation or consistency checking, can be conducted from this view by clicking one of the two possible play buttons on the right panel. The difference between these two buttons is that the right one validates the model again, based on the output *determinations* from a previous validation without resetting its *determinations*. This means invalid *choices* will not be considered any more. The other play button performs same consistency checking steps again. How the reasoning is performed will be discussed later in chapter 4.2.1.5 Model Analyser Connection.



```
[stop] {Class}
[mp3Playlist] {Class}
[MP3Player] {Attribute}
[add] {Receive Event (covered)}
[MP3Playlist] {OwnedOperation}
[add] {Class}
[MP3Player] {Operation}
    play
    play, stop
    pause, play
    play, add
    play, stop, add
    pause, play, add
[pause] {Class}
```

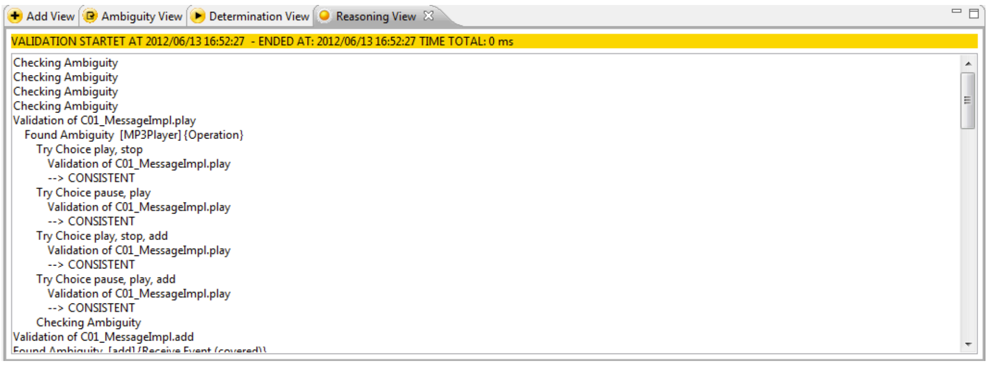
```
C01_MessageImpl.play  - CONSISTENT
C01_MessageImpl.add  - INCONSISTENT
  Determination for C01_MessageImpl.add
  Determination for C01_MessageImpl.add
    [add] {Receive Event (covered)}
      [player]
    [MP3Player] {Operation}
      [pause, play, add]
  Determination for C01_MessageImpl.add
    [add] {Receive Event (covered)}
      [player]
    [MP3Player] {Operation}
      [pause, play]
  Determination for C01_MessageImpl.add
    [add] {Receive Event (covered)}
```

Nothing In Progress.

Figure 23: The DeterminationView to generate the choices and to validate.

The ReasoningView

The *ReasoningView*, shown in Figure 24, is the fourth view of the *Ambiguitymanager* tool and simply displays the conducted validations steps in a formatted overview. Additionally, this view shows the duration of a reasoning process.



```
VALIDATION STARTET AT 2012/06/13 16:52:27  - ENDED AT: 2012/06/13 16:52:27 TIME TOTAL: 0 ms
Checking Ambiguity
Checking Ambiguity
Checking Ambiguity
Checking Ambiguity
Validation of C01_MessageImpl.play
  Found Ambiguity  [MP3Player] {Operation}
    Try Choice play, stop
      Validation of C01_MessageImpl.play
      --> CONSISTENT
    Try Choice pause, play
      Validation of C01_MessageImpl.play
      --> CONSISTENT
    Try Choice play, stop, add
      Validation of C01_MessageImpl.play
      --> CONSISTENT
    Try Choice pause, play, add
      Validation of C01_MessageImpl.play
      --> CONSISTENT
    Checking Ambiguity
Validation of C01_MessageImpl.add
Found Ambiguity  [add] {Receive Event (covered)}
```

Figure 24: The ReasoningView to see all validation steps.

4.2.1.2 Persistence

Ambiguities must be saved permanently in order to work with them after a program has been shut down. In the *Ambiguitymanager* tool ambiguities are persisted directly into a folder in the current project. The persistence as well as the feature definition are XML-driven, meaning information must be defined manually. These definitions describe information, e.g. whether a container is required to persist new *choicevalues* (e.g. class operations need a container such as a class to exist) or whether a *choicevalue* is containable (e.g. a class can or cannot be packed into another class).

The following simplified code snippet Code 7 shows such a definition for a UML class and its feature *superClass* as well as for a UML message and its feature *receiveEvent*. The feature *superClass* simply represents a possible superclass of a class. Since a class can have only one superclass, this feature can describe just one relation to another class. The first XML *attributes label*, *name*, and *opposite* are explained in 4.2.1.1 Graphical User Interface and are defined accordingly.

Additionally, there is a new *attribute* called *containable*. The designer can choose UML elements from the project as *choicevalues* for an ambiguity. New UML elements might have to be created for a single ambiguity. The fact that some UML elements cannot physically exist in a project without a container, respectively an owner element, may be a problem. An UML operation, to just mention one, cannot be alive in a project without a component that holds it in its feature *ownedOperation*. Thus, there must be an ownership relation for an operation and another element. There also exist UML elements, such as classes or interfaces, where an ownership is not required. This is a result of the fact, that there exists an ownership between the project instance and a class or interface. That means that there de facto is no need to define a certain element type to persist them.

The basic idea to solve that ownership issue is to create such a container element based on an ambiguity owner's type and ambiguity's feature. This is due to the fact that if an ambiguity for a UML element and a certain feature exist, it must be possible to take the same UML type as the ambiguity's as well as the same feature to encapsulate the newly created *choicevalue* . Thus, the element with the same type of the ambiguity's owner functioned as a helper container, respectively as a temporary owner. In the case of the feature *superClass* the required type is a UML class and classes must not be packed into another element to be considered alive in the project. Thus, the value of the *container attribute* is defined here as FALSE while the default value, respectively a no value, stands for TRUE.

If a container for a *choicevalue* must exist and the type of the selected feature has a type with which it is impossible to create a concrete UML element instance (meta types), an additional container type must be defined (called *container* in the XML definition*)*. This is the case, if an element of a feature cannot be directly user-manipulated, such as an UML callevent and its feature *operation*.

In addition, the feature of a container (in the XML definition called *featurecontainer*) has to be defined so that a *choicevalue* can be packed into it.

Another *attribute* that has not been mentioned before is called *type*. It is required if an element type of a feature is generated introspectively from a feature definition. The type can be a meta-type such as an interface; in this case creating instances from it is not allowed. Thus, a type of a *choicevalue* must be defined to create a concrete instance and to persist it into the project. Unfortunately, as of yet that type is limited by its definition and cannot be changed at runtime if another type is required in a certain case.

```
<Class literal="CLASS">
    <Feature label="Super Class" name="superClass"
     containable="FALSE" opposite="NO" />
</Class>

<Class literal="MESSAGE">
    <Feature label="Receive Event (covered)" name="receiveEvent"
     nestedfeature="covered"
     nestedtype="MESSAGE_OCCURRENCE_SPECIFICATION"
     value="LIFELINE" diagramKind="SEQUENCE_LITERAL"
     opposite="NO"/>
</Class>
```

Code 7: XML persistence definition.

Another interesting element for the persistence is an UML message. A message, as the one depicted in Figure 3, is a call between lifeline instances.

In addition to the already explained attributes *name*, *label*, *opposite*, and *container* there are additional ones called *value*, *diagramKind, nestedFeature*, and *nest-*

edType, The *value attribute* describes the type for a new added *choicevalue* and is represented by a reference literal. The *diagramKind attribute* is required if a *choicevalue* must be contained into a certain diagram

In some cases, features cannot be accessed directly due to the fact that they are nested in a super-feature. Thus, the element type of the super-feature consists of another complex type. On account of that, the required type of a *choicevalue* must be defined additionally. Due to this, the sub-feature is described by the *nestedFeature attribute* and the *nestedType attribute* for the type of the sub-feature. Hence, the element type of a selected *choicevalue* is defined and an appropriate element can be created.

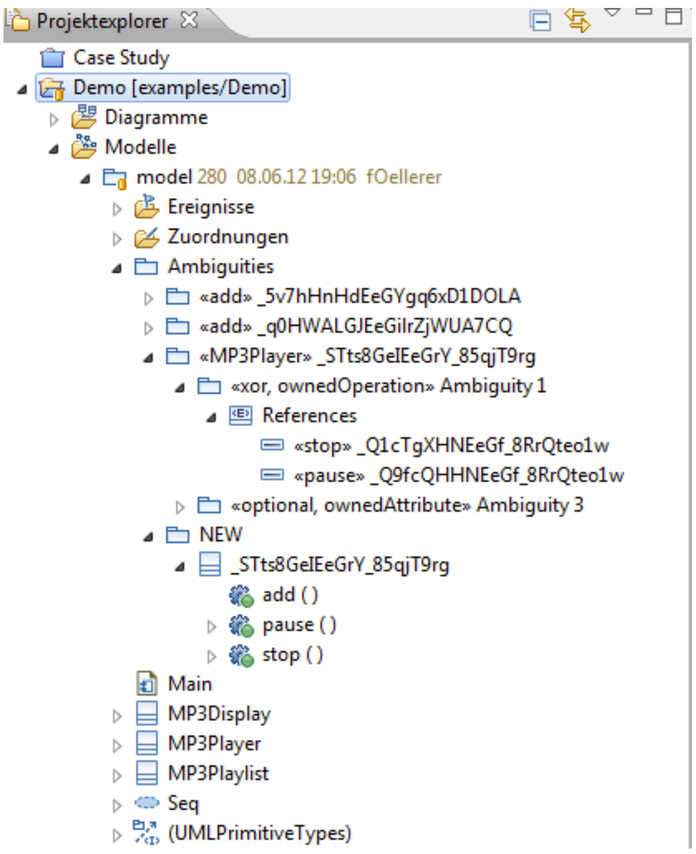
In the feature definition, however, this issue can be found in the description of an UML message and its feature *receiveEvent*. That feature does not require an element of the type receiveevent. It requires an element of the type of a receiveevent's feature named *covered*. That *covered* feature element type is the required one of the feature receiveEvent of the UML message. Accordingly, the element type of the *covered* feature is called *MESSAGE_OCCURRENCE_SPECIFICATION* and has to be taken instead of the type receiveevent.

The question, why there is a need to create concrete *choicevalues* instead of plain strings that can be parsed to create concrete instances if required, may arise. One answer is that it might be useful to manipulate *choicevalues* directly with the RSA property editor in order to change their values or to use them for other ambiguities. If the whole *choicevalue* must be parsed every time it is used to see its properties via the editor, the performance would be affected. Additionally, the selected solution is more appropriate in terms of further operations such as some reasoning processes or the resolve of a certain location *choice* and its *choicevalues*, where the concrete instance is required.

The screenshot in Figure 25 shows the persistence of ambiguities in the project explorer of the RSA. There is a folder called *Ambiguities* where all ambiguities and their *choicevalues* are stored. For each UML model element a folder is created. The folder name is a combination of a keyword (`<<MP3Player>>`) and its model element identification number (id). While the name of an UML element might be changed, an id never changes and is unique for all projects in a workspace.

The ambiguities for a model element are stored in a nested folder and named according to the user-defined ambiguity name (`Ambiguity1`). This folder has two keywords the relation kind and the feature name (`<<xor>>`, `<<ownedOpera-tion>>`). Additionally, there is another enumeration folder called *References*, containing literals of ids of *choicevalues*, which refer to created *choicevalue* elements in a folder called *NEW* or to existing *choicevalues* from a project.

The *NEW* folder contains container elements where created *choicevalue* elements are stored and *choicevalue* elements, which must not be packed into a container. To separate *choicevalues* per model element (ambiguity owner), the container is named with the id of the ambiguity owner. That results due to the possibility that *choicevalue* elements belong to different locations.



```
Projektexplorer
  Case Study
  Demo [examples/Demo]
    Diagramme
    Modelle
      model 280 08.06.12 19:06 fOellerer
        Ereignisse
        Zuordnungen
        Ambiguities
          «add» _5v7hHnHdEeGYgq6xD1DOLA
          «add» _q0HWALGJEeGilrZjWUA7CQ
          «MP3Player» _STts8GeIEeGrY_85qjT9rg
            «xor, ownedOperation» Ambiguity 1
              References
                «stop» _Q1cTgXHNEeGf_8RrQteo1w
                «pause» _Q9fcQHHNEeGf_8RrQteo1w
            «optional, ownedAttribute» Ambiguity 3
          NEW
            _STts8GeIEeGrY_85qjT9rg
              add ()
              pause ()
              stop ()
        Main
        MP3Display
        MP3Player
        MP3Playlist
        Seq
        (UMLPrimitiveTypes)
```

Figure 25: The persistence of ambiguities in the RSA project explorer.

4.2.1.3 Reverse Ambiguities Generation

The reverse ambiguity generation is quite simple. Its implementation was described in chapter 3 UML and Ambiguities. For each defined *choicevalue* a reverse ambiguity is created if the *choicevalue* has a complex type. If the EMF opposite feature cannot be used because it is missing, the defined opposite feature of the XML file is used as mentioned before. Furthermore, the only *choicevalue* for a reverse ambiguity is the owner of the origin ambiguity and its relation kind has always the value *optional*. A reverse ambiguity is treated like any other manually defined ambiguity concerning the *choice* generation or any reasoning processes. There is only one exception: modifications via the graphical user interface are not allowed as in the case of plain ambiguities.

4.2.1.4 Choice Generation

The *choice* generation is implemented as described in 3.2 The Ambiguity Concept and deals with all combinations of possible *choices* of locations. The choices are presented in a user-friendly manner in the left tree of the *DeterminationView* as depicted in Figure 23.

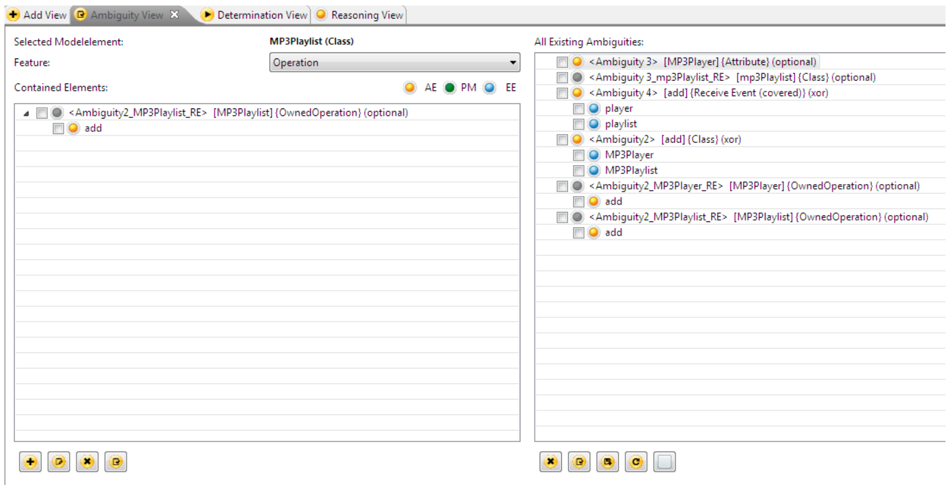
4.2.1.5 Model Analyser Connection

The requirement to connect the *Ambiguitymanager* to an existing model analyser tool is realised through the *ModelAnalyzer* plugin that introduced in 2.3.2. The ModelAnalyzer. Egyed et al. [5] developed the *ModelAnalyzer* plugin to check the consistency of models with respect to particular rules introduced in 2.2.1 Consistency Rules. The *ModelAnalyzer* uses a consistency rule as a black-box constraint and identifies affected model elements.

Concerning reasoning over UML models with ambiguities, the *ModelAnalyzer* can be used to check if a given *choice* fulfils a constraint. The analyser operates similar to the normal model analysing mechanism explained in 2.3.2. The ModelAnalyzer.

The reasoning over ambiguities and *choices* is called *Ambiguous Reasoning* (AR) [3]. This approach has been evaluated in [3]. AR is introduced in 3.3 Reasoning over UML Models with Ambiguities.

The example introduced in chapter 3.3 is implemented with the *Ambiguitymanager*. Figure 26 depicts the defined ambiguities in the *Ambiguitymanager*.



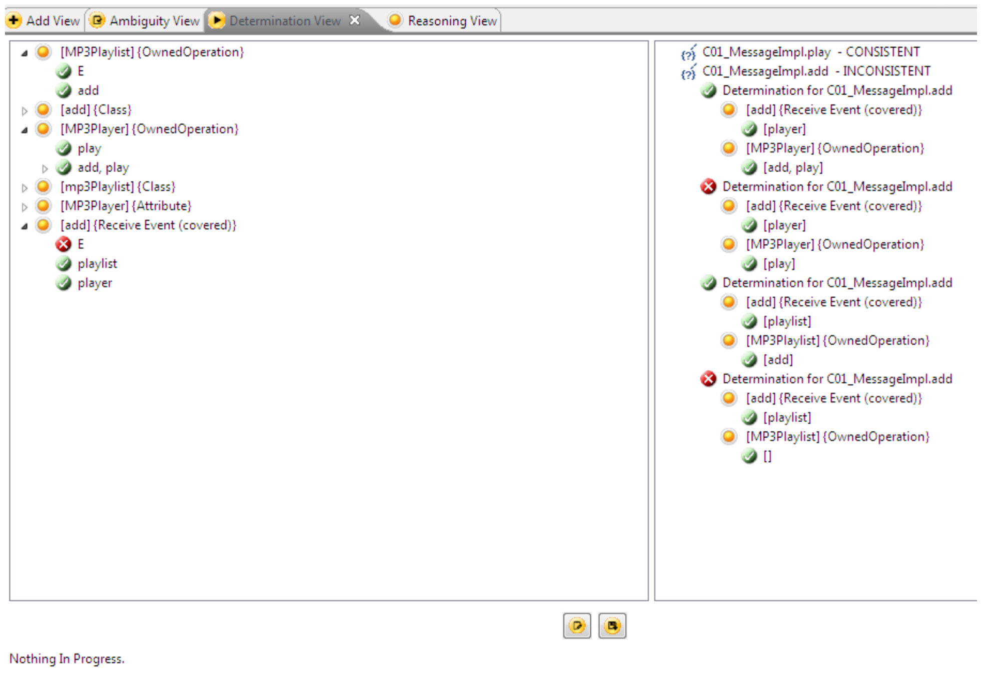
```
Selected Modelelement:    MP3Playlist (Class)
Feature:                  Operation
Contained Elements:                               AE   PM   EE

<Ambiguity2_MP3Playlist_RE>  [MP3Playlist] {OwnedOperation} (optional)
    add

All Existing Ambiguities:
<Ambiguity 3>  [MP3Player] {Attribute} (optional)
<Ambiguity 3_mp3Playlist_RE>  [mp3Playlist] {Class} (optional)
<Ambiguity 4>  [add] {Receive Event (covered)} (xor)
    player
    playlist
<Ambiguity2>  [add] {Class} (xor)
    MP3Player
    MP3Playlist
<Ambiguity2_MP3Player_RE>  [MP3Player] {OwnedOperation} (optional)
    add
<Ambiguity2_MP3Playlist_RE>  [MP3Playlist] {OwnedOperation} (optional)
    add
```

Figure 26: Visualisation of ambiguities.

Additionally, Figure 27 depicts the *choice* generation (left) and the *determinations* (right) of the example implemented with the *Ambiguitymanager*.



```
[MP3Playlist] {OwnedOperation}
    E
    add
[add] {Class}
[MP3Player] {OwnedOperation}
    play
    add, play
[mp3Playlist] {Class}
[MP3Player] {Attribute}
[add] {Receive Event (covered)}
    E
    playlist
    player

C01_MessageImpl.play  - CONSISTENT
C01_MessageImpl.add  - INCONSISTENT
  Determination for C01_MessageImpl.add
      [add] {Receive Event (covered)}
          [player]
      [MP3Player] {OwnedOperation}
          [add, play]
  Determination for C01_MessageImpl.add
      [add] {Receive Event (covered)}
          [player]
      [MP3Player] {OwnedOperation}
          [play]
  Determination for C01_MessageImpl.add
      [add] {Receive Event (covered)}
          [playlist]
      [MP3Playlist] {OwnedOperation}
          [add]
  Determination for C01_MessageImpl.add
      [add] {Receive Event (covered)}
          [playlist]
      [MP3Playlist] {OwnedOperation}
          []

Nothing In Progress.
```

Figure 27: Visualisation of choices and determinations.

4.2.2 Limitations

While developing the *Ambiguitymanager* limitations in terms of expressions restrictions and scalability problems were encountered. The first means in detail, that a grouping mechanism of ambiguities, which depend on one another or must always be selected together does not yet exist. This issue will be discussed in 6 Problems and Limitations.

Furthermore, a problem with the RSA occurs by generating *choices* for a location. The approach of the *choice* generation is not scalable in terms of the combinations of all ambiguity *choices* and their visualisation in the *DeterminationView*. The following example in Table 3 provides an overview of the difficulty[3]:

```
3 ambiguities (relation kind optional)

10 choicevalues per ambiguity

2^10 choices per ambiguity = 1.024 choices

Combination of all 1.024^3 choices

= 1.024*1.024*1.024

= 1.073.741.824 choices per location (+ 3.072 ambiguity choices)
```

Table 3: Determination of choices.

Thus, if there are three ambiguities for a location with 10 *choicevalues*, there are 1.073.741.824 *choices* and even more *choicevalues*, which must be visualised via a graphical tree. At the moment, this is solved by a configuration of the graphical tree, meaning that its items are closed per default. Thus, all *choices* and *choicevalues* have to be visualised only once the depending parent item is opened.

[3] That calculation does not consider duplicates of *choices*. *Choices* are here treated as unique elements and are not equals if they enfold the same *choicevalues*.

5 Case Study

The Ambiguity Concept discussed in 3.2 The Ambiguity Concept provides an approach to express different design decisions of UML models and their elements. The running example introduced in 3.1 Illustration and Running Example consists of 12 UML model elements and is reproduced in the RSA with the assistance of the *Ambiguitymanager*.

To prove that the Ambiguity Concept and the *Ambiguitymanager* can handle a wider range of model elements and their ambiguities, a huge case study has been implemented. This case study was based on a requirements document for the Barbados Crash Management System Product Line (bCMS-SPL) from Istoan et al. [37].

Istoan et al. [24] modelled based on the bCMS-SPL requirements document a product line and a *reference variant* in an object-oriented way with the UML. The *reference variant* consists of functionalities, which can be found in any product of the bCMS-SPL. Thus, they are mandatory for any derived product. All models created for the *reference variant* can be used as a basis for defined variation points and additional behavioural and structural properties.

Based on the resulting models, the Ambiguity Concept is used to express the variation points of the product line. The case study consists of hundreds of model elements and their class, state machine, and sequence diagrams. While the requirements document of the case study also encloses non-functional requirements, this thesis will focus on the functional aspects of the bCMS-SPL.

The bCMS-SPL describes a crisis management system to handle accidents on roadways in terms of the coordination of firemen and policemen. Policemen and firemen have different responsibilities and their concurrently executed tasks must be coordinated in an effective and efficient way. [24]

To express variability with the Ambiguity Concept understanding the entire background and intension of a system's behaviour and structure is not mandatory. Thus, the expressed ambiguities are based on textual descriptions, the results of definitions of variation points, and not on the requirements document from Istoan et al. [37].

This chapter provides an overview about the main components of the focused case study. The derivation of ambiguities from textual descriptions will be pointed out.

In addition, the solution found in the focused case study will be compared to the realisation via the Ambiguity Concept. A section of resulting *choices* and *determinations* of the reasoning process will be presented. Finally, strengths, weaknesses, problems, and limitations concerning the Ambiguity Concept and in terms of reasoning over a huge number of model elements and ambiguities will be discussed.

5.1 Overview

In this subchapter an overview about system components such as the domain model of the bCMS-SPL will be provided. First, the domain model will be presented and key elements will be introduced. In a second step, the depending feature model and its variation points will be elicited.

5.1.1 Domain Model

In this section the domain model and its key elements will be presented. The domain model includes system components and physical elements. All components are modelled accordingly with the RSA. Physical elements describe components, which are outside the system and interact with the system in a certain manner. Physical elements can be fire trucks, police cars, or victims, which have to be rescued.

The major components of the case study are two human actors, the Fire Station Coordinator (FSCoordinator) on the fire station side and the Police Station Coordinator (PSCoordinator) on the police station side. Both of them stand for the interaction of human beings with the crisis management system. Additionally, the FSCSystem class and the PSCSystem class build the centre of the product line. The FSCSystem manages interactions between the FSCoordinator and the PSCSystem. The PSCSystem class manages interactions between the PSCoordinator and the FSCSystem.

Since, as mentioned before, understanding the background and intension of a system's behaviour and structure is not mandatory to express variability with the Ambiguity Concept, there will be no further explanation of the classes and actors.

The case study captured key functional scenarios based on the requirements in [37]. The functional requirements are described via sequence diagrams and show the interactions between system and physical components such as the FSCSystem and the FSCoordinator. The scenarios consist of seven main scenarios and additional, alternative, and exceptional behaviour. Alternative and exceptional behaviour generally can affect all scenarios such as if a connection between components gets lost. The following scenarios are taken from Istoan et al. [24] and describe the main scenarios of the bCMS-SPL.

1. *PSCSystem and FSCSystem establish communication and identification of coordinators.*
2. *PSCSystem and FSCSystem exchange crisis details.*
3. *PSCSystem and FSCSystem develop a coordinated route plan in a timely fashion for number of vehicles to be deployed to specific locations with respective ETAs (estimated time of arrival).*
    - *3.1. PSCSystem and FSCSystem state their respective number of fire trucks and police vehicle to deploy.*
    - *3.2. PSCSystem proposes one route for fire trucks and one route for police vehicles to reach crisis site.*
    - *3.3. FSCSystem agrees to route.*
4. *PSCSystem and FSCSystem communicate to each other that their respective vehicles have been dispatched according to plan (per vehicle).*
5. *PSCSystem and FSCSystem communicate to each other their arrival (per vehicle) at targeted locations.*
6. *PSCSystem and FSCSystem communicate to each other completion (per vehicle) of their respective objectives.*
7. *PSCSystem and FSCSystem agree to close the crisis.*

For each scenario there is a depending detailed sequence diagram. By implementing the case study, all sequence diagrams are modelled accordingly with the RSA. Thus, there are eight sequence diagrams, which describe seven main scenarios and one exceptional behaviour of the overall domain model. Existing loop conditions in sequence diagrams are then omitted in the RSA. This is due to the fact, that loop conditions do not have benefit in terms of expressing variation points with the Ambiguity Concept. The key elements of the bCMS-SPL, respectively the domain model, is represented by an UML class diagram and depicted in Figure 28.

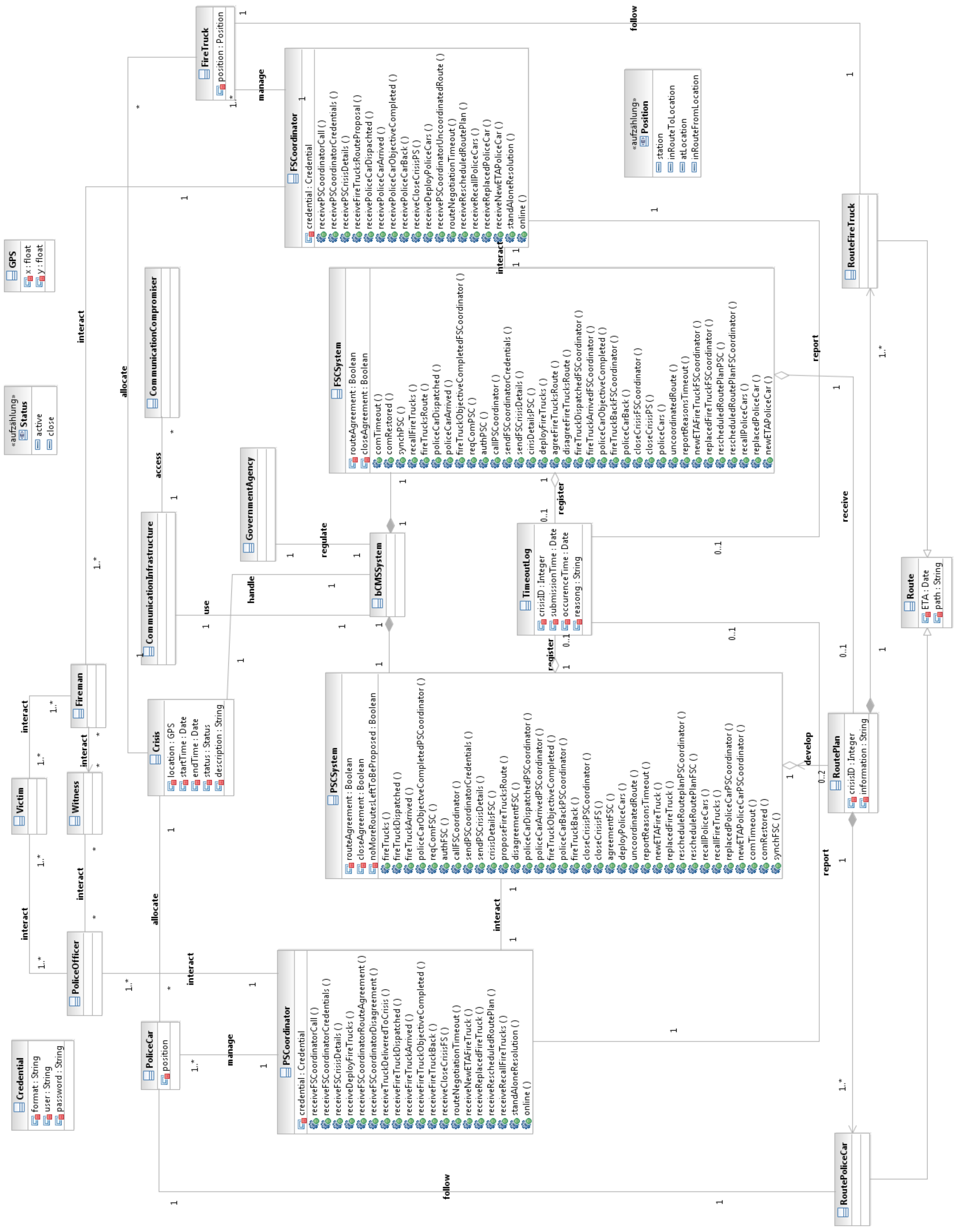


Figure 28: Domain model of the bCMS-SPL.

Furthermore, there are six state machine diagrams for the PSCoordinator, FSCoordinator, PSCSystem, and the FSCSystem. While four of them describe in detail all possible states of these components, two additional state machine diagrams concerning the connectivity of the PSCSystem and the FSCSystem are defined. The additional two diagrams describe the availability of both systems in terms of a communication channel, concretely, whether they are in a standalone mode or in a collaborative crisis management mode. [24]

5.1.2 Feature Model

A feature model describes features of a product line and depicts its variation points as discussed in 2.4 Related Work. The implementation of the bCMS-SPL case study will focus on the functional features or rather on variation points.

Figure 29 provides an overview about the six features of the focused product line. Four of them represent variation points; their textual description from [24] will be used to express their requirements via the Ambiguity Concept. The four variation points are *P&F station multiplicity*, *Vehicles management, Vehicles management communication protocol*, and *Crisis multiplicity.* The domain model introduced in 5.1.1 Domain Model and its depending class, sequence, and state machine diagrams covers the requirements of two features, existing in every derived product such as *Communication establishment* and *Coordinator identification.* Due to this, they are not translated into ambiguities and will not be discussed any further.

Each of the four variation points and additional required class, sequence and state machine diagrams are depicted with the RSA and their textual description are transformed into ambiguities. Thus, there are 136 ambiguities, 75 reverse ambiguities, 38,344 resulting *choices*, and 16,824,396 *choicevalues*.

In the next subchapter, one of the four variation points, respectively its feature, are chosen to present it in terms of the Ambiguity Concept. The variation point *Vehicles management* was chosen as a result of its complexity, i.e. that it comprises *require* statements between features and their major complexity.

While there are a huge number of textual descriptions in terms of additional requirements resulting from that variation point, only selected ones and the depending UML diagrams will be considered in this thesis.

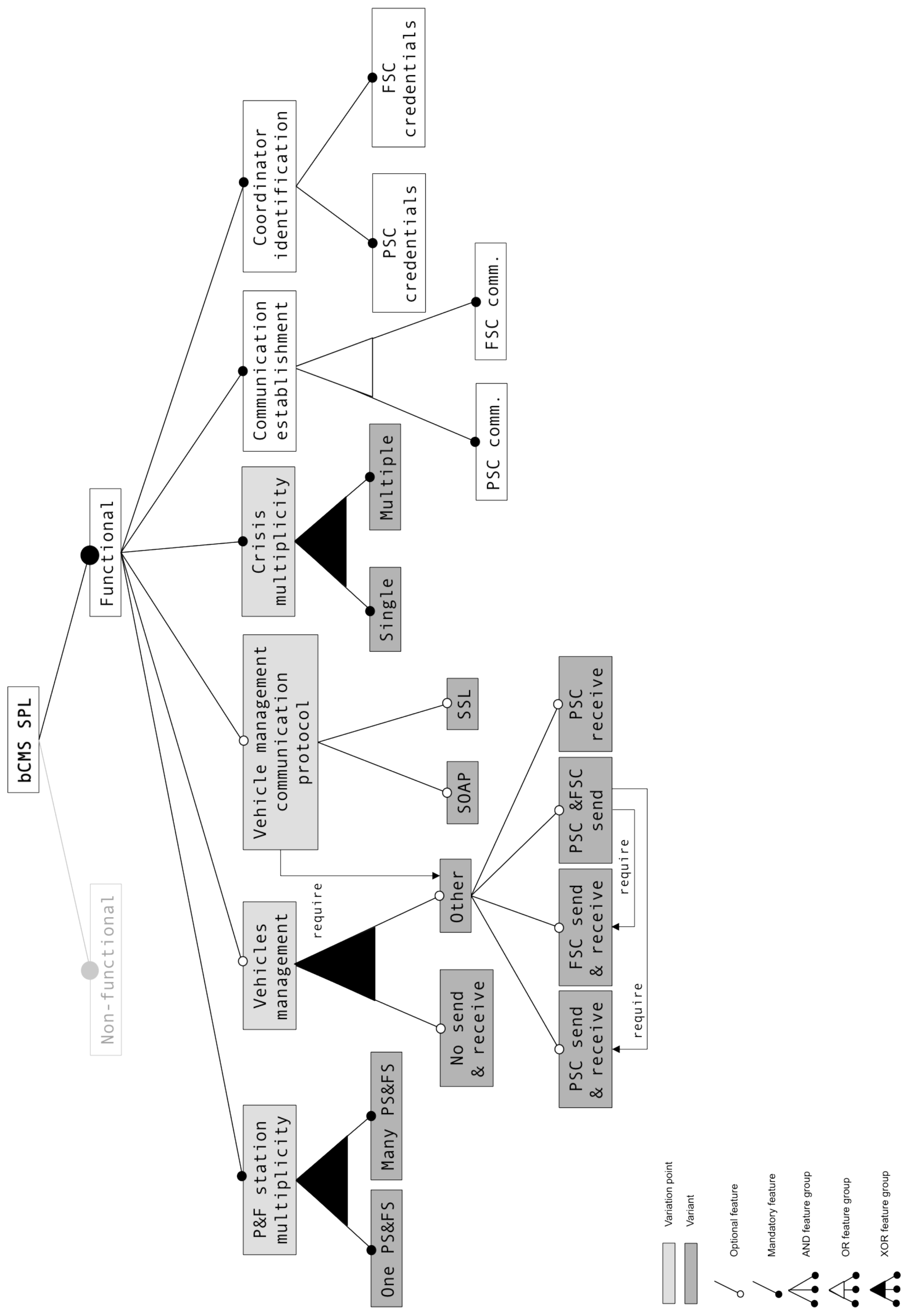


Figure 29: Feature model of the bCMS-SPL.

5.2 Implementation and Adaption

In this subchapter the previously selected variation point *Vehicles management* is expressed through the Ambiguity Concept. In addition, problems and limitations concerning the Ambiguity Concept will be discussed. Furthermore, a comprehension between the approach to express variability in [24] and the Ambiguity Concept will be outlined. Additionally, limitations concerning the Ambiguity Concept in terms of its restricted expression language concerning textual feature descriptions will be pointed out. The solution discovered in the course of the focused case study will be compared to the realisation with the Ambiguity Concept. Additionally, a section of resulting *choices* and *determinations* of a reasoning process will be presented. Finally, advantages and disadvantages of the *Ambiguitymanager* and of reasoning over a huge number of model elements and ambiguities will be discussed.

5.2.1 Vehicles Management Variation Point

The textual description of the variation point *Vehicle management* provides a simple usage of the Ambiguity Concept. There is no changes if the sub-feature *No send & receive* is selected because it is already covered in the reference variant. Thus, this thesis will focus on the sub-feature *Other* and its feature group.

First, structural changes and depending class diagrams will be discussed. In a second step, aspects concerning behavioural changes in sequence and state machine diagrams will be expressed through the Ambiguity Concept.

Please note that ambiguities mentioned in the following sections act as representatives for similar ambiguities. Such similar ambiguities will not be pointed out here, but have been defined in the RSA.

5.2.1.1 Structural Changes

The following six sentences concern the structure of the bCMS-SPL and can easily expressed through ambiguities.

1. Create a new class called DispatchService with certain operations.

To solve this requirement, an ambiguity is created for the model element *bCMSProject* and its feature *ownedElement*. The *bCMSProject* represents the RSA modelling project that comprises the whole case study. The possible *choicevalue* is a new element called DispatchService and the relation kind is *optional*. The required operations are added to the DispatchService.

The ambiguity is formulated as:

```
A1 = (bCMSProject+ownedElement, opt, {DispatchService})
```

2. Add a relation to the PSCSystem and FSCSystem class.

Due to the fact that DispatchService does not yet exists in a class diagram, it has to be inserted either into a new or the existing domain model to add relations to it. Thus, associations from PSCSystem and FSCSystem to the DispatchService must be created.

In addition, two ambiguities must be defined to express that created relations are optional elements. Please note that class attributes and associations are considered as equals in the EMF framework.

The ambiguities are formulated as:

```
A2 = (PSCSystem+ownedAttribut, opt,
{associationToDispatchService})

A3 = (FSCSystem+ownedAttribute, opt,
{associationToDispatchService})
```

3. Connect DispatchService with PoliceCar, FireTruck, and CitizenVehicle class.

According to the previous sentence, associations between PoliceCar, FireTruck, and CitizenVehicle directed to the DispatchService must be created.

The ambiguities are formulated as:

```
A4 = (PoliceCar+ownedAttribute, opt,
{associationToDispatchService})

A5 = (FireTruck+ownedAttribute, opt,
{associationToDispatchService})

A6 = (CitizenVehicle+ownedAttribute, opt,
{associationToDispatchService})
```

Please note that it makes no difference whether ambiguities for DispatchService or for those three classes are created in that way. This is due to the fact, that generated reverse ambiguities of the two possibilities cover the other definition as well.

4. Tag DispatchService as optional due to the fact that there exists no way in the UML to express optional UML elements.

This sentence is already expressed by the first sentence and its ambiguity: bCMSProject either owns DispatchService or does not. Thus, DispatchService can be considered as an optional element.

5. Connect the DispatchService with the sub-feature *Other* of *Vehicle Management* to express that the class only exists if *Other* is selected.

The current sentence cannot be expressed via the Ambiguity Concept due to the missing possibility to add *require* statements of different UML elements. If there exists such a mechanism, an ambiguity could be created with a *requires* relation and the two *choicevalues* DispatchService and all elements resulting of the feature *Other*. That issue will be discussed in 6.3.2 Grouping Mechanism of Choices.

6. Tag every operation of DispatchService as optional.

In order to solve this requirement, ambiguities are created for DispatchService and its feature *ownedOperation.* The possible *choicevalues* consists of each existing operation in the DispatchService and the relation kind is *optional.* Only two of them will be mentioned here.

The ambiguity is formulated as:

```
        A7 = (DispatchService+ownedOperation, opt,
        {policeCarDispatchOrder(), fireTruckDispatchedOrder()})
```

5.2.1.2 Behavioural Changes

This subchapter deals with behavioural changes in the domain model due to the variation point *Vehicle management*. While the sub-feature *No send & receive* is selected per default and does not affect the structural and behavioural properties of the bCMS-SPL, the feature group *Other* does. Due to this fact, this section describes possible expressions of behavioural changes with the Ambiguity Concept.

In [24] behavioural aspects are modelled with sequence and state machine diagrams. The authors created a lot of new sequence diagrams with the UML *optional interaction fragment.* This element allows a user to add optional parts to a sequence diagram, which are only present if a defined guard condition is fulfilled. Accordingly, if the condition is evaluated to false, it is not present in the sequence diagram.

For each sub-feature an *optional interaction fragment* is created. The guard condition of each fragment can be evaluated to true, if the depending sub-feature is selected. For the sub-feature *PSC send & receive* the guard condition is expressed in the following way:

```
"PSC send&receive" = selected
```

To express this variation point in terms of the sequence diagrams with the Ambiguity Concept, all *optional interaction fragments* are created as well. There also exists the option to create a new fragment outside the sequence diagram of the reference variant. That means in detail, to add an ambiguity for the model element *DomainSequence* and its feature *ownedElement* and to create all other lifelines, calls between them, guard conditions, etc. without a visualisation as well. Thus, it would be very hard to gain an overview about a model, respectively a sequence diagram, without seeing the components and their relations.

However, there is an additional way of expressing optional behaviour without modelling the whole parts without visualisation. As a possibility, the designer can separate the fragments via sub-feature into sequence diagrams and add ambiguities, which describe the optionality in terms of the whole project.

Such an ambiguity is formulated as:

```
A8 = (bCMSProject+ownedElement, opt, {PSCsendReceiveSequence})
```

On account of this, the optionality for every other sub-feature of the *Other* feature group can be expressed in this way. But even if the optionality can be expressed, the sequence diagrams must be created as well. Due to this, the Ambiguity Concept does not save any effort in terms of creating sequence diagrams.

Furthermore, state machine diagrams are created. They describe possible states and are based on the state machine diagrams created for the *reference variant*. The authors of the focused case study modelled the whole state machine diagrams again and modified affected elements. Modified elements here are transitions (named with an operation of the component depending on the state machine) between states and return-values (UML *OpaqueBehavoir* interface) after a transition was conducted. While all state machine diagrams can be taken from the models of the *reverence variant*, a diagram for the DispatchService must be newly created. This is due to the fact, that this class does not exist in the reference variant and thus, there is no state machine diagram yet.

To express the mentioned modifications with the Ambiguity Concept, there is no need to model the entire diagrams again. Thus, the following examples show how easily those changes can be expressed with the ambiguity approach. In the case study there are no real textual descriptions for that part, but they can be derived via the comparison of diagrams of the *reference variant* and the created ones. The chosen state machine describes states for the PSCSystem and there will be two representative examples discussed. The first explains how to express a transition of a state and the second one a return value.

1. Change the body condition of the state named with S.5.1 from *receivePolice-CarArrivedAtCrisis AND numArrivedPoliceCars++* to *policeCarsArrived AND numArrivedPoliceCars++*

The ambiguity can be formulated as:

```
A9 = (policeCarsArrived AND numArrivedPoliceCars++ + body, xor,
             {receivePoliceCarArrivedAtCrisis AND
              numArrivedPoliceCars++,
             policeCarsArrived AND numArrivedPoliceCars++})
```

This ambiguity says that for the model element *policeCarsArrived AND numAr-rivedPoliceCars++* (an instance of a Class that implements the OpaqueBehavoir Interface) and its feature body the possibility exists that its describing return-value might change into a newly created value named with *receivePoliceCarArrivedA-tCrisis AND numArrivedPoliceCars++*. The relation kind with value *xor* results in the possibility that such a return value can consist of more than one. Thus, both cannot exist together and the *xor* value expresses that mutually exclusive relation.

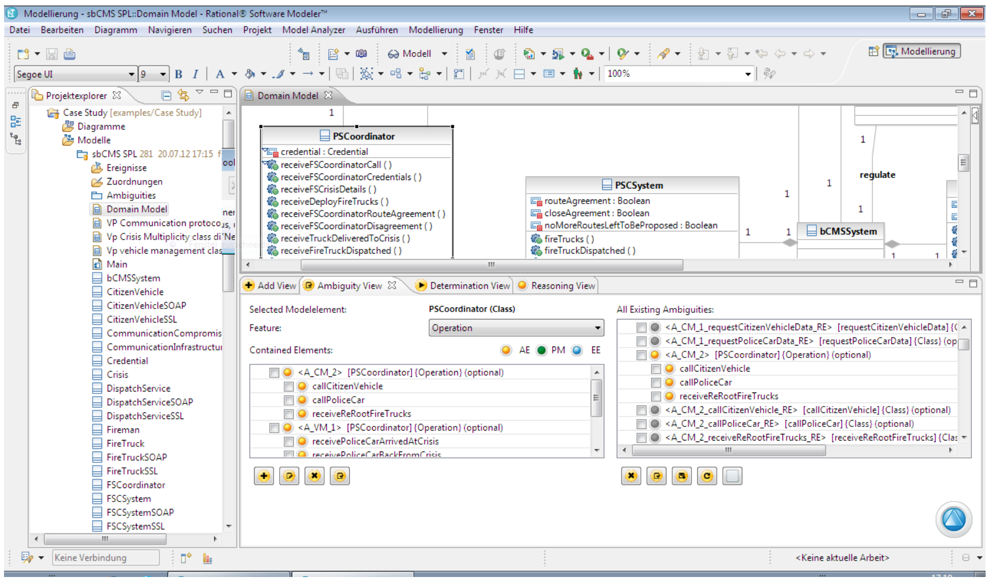
2. Change the transition name of the state named S.5.1 from *police-CarArrivedPSCoordinator* to *receivePoliceCarArrivedAtCrisis*.

The ambiguity can be formulated as:

```
A10 = (policeCarArrivedPSCoordinator+name, opt,
{receivePoliceCarArrivedAtCrisis})
```

The ambiguity expresses that there exists a transition named *police-CarArrivedPSCoordinator* or *receivePoliceCarArrivedAtCrisis*. The optional relation kind can be used, since the feature *name* has the multiplicity exactly one and thus only one of them can be selected at the same time. There is no need to declare a mutually exclusive relation kind.

Finally, Figure 30 depicts a screenshot of the case study and defined ambiguities in the *Ambiguitymanager*.



Modellierung - sbCMS SPL::Domain Model - Rational® Software Modeler™
Datei Bearbeiten Diagramm Navigieren Suchen Projekt Model Analyzer Ausführen Modellierung Fenster Hilfe
Projektexplorer
Case Study [examples/Case Study]
Diagramme
Modelle
sbCMS SPL 281 20.07.12 17:15
Ereignisse
Zuordnungen
Ambiguities
Domain Model
VP Communication protocol
Vp Crisis Multiplicity class
Vp vehicle management class
Main
bCMSSystem
CitizenVehicle
CitizenVehicleSOAP
CitizenVehicleSSL
CommunicationCompromis
CommunicationInfrastructu
Credential
Crisis
DispatchService
DispatchServiceSOAP
DispatchServiceSSL
Fireman
FireTruck
FireTruckSOAP
FireTruckSSL
FSCoordinator
FSCSystem
FSCSystemSOAP
FSCSystemSSL
Domain Model
PSCoordinator
credential : Credential
receiveFSCoordinatorCall ( )
receiveFSCoordinatorCredentials ( )
receiveFSCrisisDetails ( )
receiveDeployFireTrucks ( )
receiveFSCoordinatorRouteAgreement ( )
receiveFSCoordinatorDisagreement ( )
receiveTruckDeliveredToCrisis ( )
receiveFireTruckDispatched ( )
PSCSystem
routeAgreement : Boolean
closeAgreement : Boolean
noMoreRoutesLeftToBeProposed : Boolean
fireTrucks ( )
fireTruckDispatched ( )
bCMSSystem
regulate
Add View
Ambiguity View
Determination View
Reasoning View
Selected Modelelement:
PSCoordinator (Class)
Feature:
Operation
Contained Elements:
AE
PM
EE
<A_CM_2> [PSCoordinator] (Operation) (optional)
callCitizenVehicle
callPoliceCar
receiveReRootFireTrucks
<A_VM_1> [PSCoordinator] (Operation) (optional)
receivePoliceCarArrivedAtCrisis
All Existing Ambiguities:
<A_CM_1_requestCitizenVehicleData_RE> [requestCitizenVehicleData]
<A_CM_1_requestPoliceCarData_RE> [requestPoliceCarData] (Class) (op
<A_CM_2> [PSCoordinator] (Operation) (optional)
callCitizenVehicle
callPoliceCar
receiveReRootFireTrucks
<A_CM_2_callCitizenVehicle_RE> [callCitizenVehicle] (Class) (optional)
<A_CM_2_callPoliceCar_RE> [callPoliceCar] (Class) (optional)
<A_CM_2_receiveReRootFireTrucks_RE> [receiveReRootFireTrucks] (Clas
Keine Verbindung
<Keine aktuelle Arbeit>

Figure 30: Implementation of the case study with the Ambiguitymanager.

5.2.2 Resulting Choices

The resulting *choices* from all ambiguities are generated as discussed in 3.2.1 Relations between Ambiguities. In this section, the bCMSProject packet and its *ownedElement* feature is chosen to demonstrate a *choice* generation for the case study.

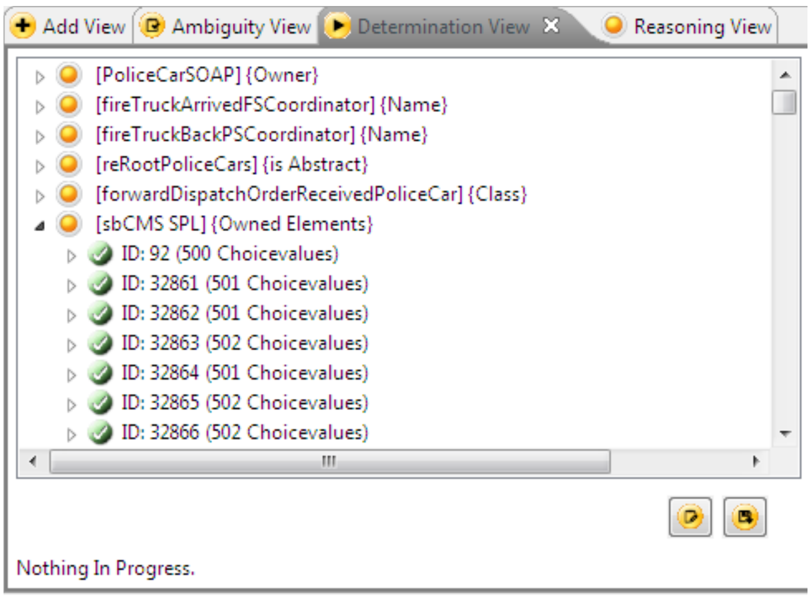
The following ambiguity expresses the optional DisplayService class as explained in the previous section:

```
A1 = bCMSProject+ownedElement → {(opt, {DispatchService})}
```

Due to the fact that there are a huge number of mandatory model elements in the bCMSProject *ownedElement* feature it goes beyond the scope of this thesis to mention all of them in the *choices*. They combined and called *M* for mandatory values.

This results in two *choices*: the first enfolds *M* and the second consists of *M* and the DisplayService class.

When implementing the case study there are a huge number of optional elements, which would result in a lot of more *choices* than the two mentioned above. Figure 31 shows a screenshot of all generated *choices* for the bCMSProject and its owned elements with the *Ambiguitymanager*.



[PoliceCarSOAP] {Owner}
[fireTruckArrivedFSCoordinator] {Name}
[fireTruckBackPSCoordinator] {Name}
[reRootPoliceCars] {is Abstract}
[forwardDispatchOrderReceivedPoliceCar] {Class}
[sbCMS SPL] {Owned Elements}
ID: 92 (500 Choicevalues)
ID: 32861 (501 Choicevalues)
ID: 32862 (501 Choicevalues)
ID: 32863 (502 Choicevalues)
ID: 32864 (501 Choicevalues)
ID: 32865 (502 Choicevalues)
ID: 32866 (502 Choicevalues)

Nothing In Progress.

Figure 31: Case study choice generation.

5.2.3 Comparison of Approaches

The implementation of the focused case study with the Ambiguity Concept illustrates its strengths and weaknesses. The following section offers a comparison of the approach used in the case study and the Ambiguity Concept.

The case study focuses on expressing variability of SPL that can be considered as the expression of design decisions for all products of a product line. The Ambiguity Concept tries to handle different design decisions as well, even if it is not yet used for product lines.

The Ambiguity Concept has the advantage that it offers the possibility to define optional elements instead of re-modelling whole model parts depending on a decision. The approach of Gomaa [2] (mentioned in 2.4 Related Work) is similar to the one used in the case study. However, Gomaa also defines optional keywords to tag elements as optional depending on a selected configuration for a product. Istoan et al. [24] define optional keywords to express variability aspects as well. In the Ambiguity Concept, optional elements result automatically from ambiguity definitions, respectively from the *choice* generation.

On account of this, *choices*, which are the results of ambiguities and existing values, can be compared with valid or invalid configurations of a derived SPL product. The former must satisfy ambiguity definitions concerning valid combinations of *choicevalues*. However, the latter is concerned with the satisfiability of defined feature models. Thus, a huge difference between the case study approach and the Ambiguity Concept is, that resulting *choices* for a location are generated automatically and must not configured by a designer (as they would have to be in a product configuration process).

Another advantage of the Ambiguity Concept is that one can select particular model elements of a complex model and add certain ambiguities instead having to re-model the entire diagrams again. This is the case in 5.2.1.2 Behavioural Changes and mentioned in state machine diagrams by adding ambiguities for transitions and return values (body).

A weakness of the Ambiguity Concept is that as of yet no mechanism for elements, which must be selected together such as those depending on a particular feature, exists. This issue can be resolved by adding a new relation kind that represents a kind of *mandatory dependency* expression.

In terms of completely new defined sequence diagrams for a variation point, as mentioned in 5.2.1.2 Behavioural Changes, the Ambiguity Concept does not yet offer any benefit in terms of modelling fewer aspects again. Due to a missing visualisation of created lifelines, calls between them, guard conditions, etc. creating an overview about a model, respectively a sequence diagram without seeing the components and their relations, would be very hard.

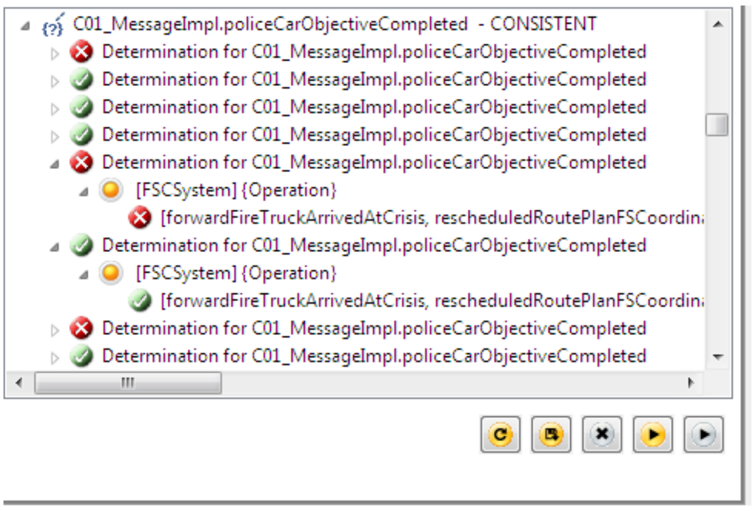
In conclusion, the implementation of a larger case study shows aspects of the Ambiguity Concept, which did not emerge in smaller modelling projects. Some of them result from the reasoning process or the huge numbers of generated *choices* and will be discussed in the next section.

5.3 Reasoning

This subchapter deals with the evaluation and Ambiguous Reasoning output, so called *determinations*, of the case study. Due to the huge number of model elements, defined ambiguities, and resulting *choices* the reasoning process is reduced to a single consistency rule. Furthermore, the first rule in Table 1 defines that the name of a message call in a sequence diagram must be an operation in the receiver's class definition. This rule is used to evaluate the case study. The reasoning process determines which *choicvalues* can be selected together and which ones represent an invalid combination in terms of the consistency rule. That kind of combination is called *determination*; it was introduced in 4.2.1.5 Model Analyser Connection.

5.3.1 Determinations

A *determination* represents a selection of an ambiguity and one of its *choices*. The case study comprises a huge number of message calls in sequence diagrams. Figure 32 shows a screenshot of all resulting *determinations* of the case study.



```
C01_MessageImpl.policeCarObjectiveCompleted - CONSISTENT
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
    [FSCSystem] {Operation}
      [forwardFireTruckArrivedAtCrisis, rescheduledRoutePlanFSCoordin
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
    [FSCSystem] {Operation}
      [forwardFireTruckArrivedAtCrisis, rescheduledRoutePlanFSCoordin
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
  Determination for C01_MessageImpl.policeCarObjectiveCompleted
```

Figure 32: Case study determinations.

5.3.2 Problems

By implementing the Ambiguity Concept and reasoning over a huge number of ambiguous and non-ambiguous model elements, some problems occurred in terms of the performance and scalability.

Reasoning over a huge number of model elements, constraints and *choices* can lead to performance problems. This results in a huge number of model elements that must be encountered during the evaluation. Thus, the evaluation of the case study with only one rule takes on average 38,000 milliseconds. The *choice* generation time separated takes on average 5,100 milliseconds; this can be considered as an acceptable result, but certainly, depends on the used computer performance.

6 Problems and Limitations

In this chapter, some limitations and problems, which occurred during the implementation of the case study and the adaption of the Ambiguity Concept with the *Ambiguitymanager* will be pointed out. Additionally, some general issues and possible extensions concerning the Ambiguity Concept will be discussed.

6.1 Storage and Encapsulated Features

In theory it is not difficult to create new elements for an ambiguity and to play around with them. However, in practice it has been shown that it is difficult to create and to store those elements in an appropriate way. In addition, certain features of UML elements are more complex and structured differently.

6.1.1 Storage of Elements

Concerning the storage of elements as described in 4.2.1.2 Persistence, there are some restrictions in terms of the creation of UML elements in the RSA. Thus, some UML elements cannot exist physically in a project without a container, respectively an owner element. Hence, an UML operation cannot be alive in a project without a component that holds it in its feature *ownedOperation*. Thus, an ownership relation for an operation and another element must exist. Since this must be considered for any UML element, there is no generic approach to solve that issue. Thus, for almost every UML element a certain implementation of its persistence has to exist. Unfortunately, the type of the owner element cannot be derived automatically from the UML specification. Since the specification declares meta-types of elements the concrete type is not available.

Concerning the implementation of the case study the following ambiguity shows the problem in detail:

```
A7 = (DispatchService+ownedOperation, opt,
{policeCarDispatchOrder(), fireTruckDispatchedOrder()})
```

The DispatchService has two optional operations (`policeCarDispatchOrder()`, `fireTruckDispatchedOrder())`. To solve this problem, a container must be created based on the feature description in the XML file of Code 6 for the feature *ownedOperation* of a class since the operations do not exists in the project yet.

6.1.2 Encapsulated Features

On account of this, there are features of the UML feature specification, which are encapsulated in other features. This results in more effort concerning a specific implementation of UML features as described in 4.2.1.2 Persistence.

However, in the feature definition this issue can be found in the description of an UML message and its feature *receiveEvent*. The *receiveEvent* is not accessible directly, but is represented by its feature *covered*. Accordingly, the element type of that sub-feature is called *MESSAGE_OCCURRENCE_SPECIFICATION*.

6.2 Optional Elements

Another interesting aspect surfaces if an ambiguity declares that a *choicevalue* must be deleted if a specific *choice* is selected. Currently, the designer can define an ambiguity in that the UML element of a *choicevalue* is treated as optional concerning its parent element (e.g. a package is the parent of a class) and the particular feature. In the *Ambiguitymanager* an optional element actually exists in both cases in the project independent from a selected *choice*. Thus, the realization of a *choice* without that optional element must perform the physically deletion process of it and cannot be used in other defined ambiguities and their *choices*. Furthermore, the *choices* resulting from other ambiguity must contain *nullpointers* or a unique reference element to symbolise a non-existing element. That issue shows that model elements do have a relationship among one another.

6.3 Implicit and Explicit Dependencies

The Ambiguity Concept and the *Ambiguitymanager* as well are restricted in terms of the definitions of dependencies between any kinds of elements. Dependencies can be implicit or explicit. Explicit dependencies can be detected by a reasoner and can be based on constraints. However, implicit dependencies can be invalid concerning a specific intention or idea of a designer. Those dependencies cannot be found via the *Ambiguitymanager*. Even if this is not possible yet, the connected reasoner evaluates a model based on defined rules. Thus, rules or constraints describing non-formal knowledge and restrictions can be defined as well. Due to this, the reasoner can be used to detect those inconsistencies.

For a better understanding, the following sections provide concrete examples of such dependencies.

6.3.1 Grouping Mechanism Choicevalues

The *Ambiguitymanager* reaches its limits if there is need for *choicevalues*, which contain a group of elements. The following example illustrates that issue with the aid of the mp3 player scenario.

There is a need to create an ambiguity for the whole project of the mp3 player scenario (called here *project*) and all its containing elements (called here *ownedElement*). Thus, the location consists of the element `project` and its feature `ownedElement`. The idea is to express the fact that some groups of elements are optional and mutually exclusive. Optional elements are the MP3Playlist and the association (called here *connection*) to the Mp3Player, because they are only alive if the first variant is selected. Furthermore, the sequence diagram depicted in Figure 9 contains two optional elements as well. The former represents the Lifeline (called here *lifeline*) of the MP3Playlist, the latter the message call (called here *call*) from the MP3Player to the MP3Playlist.

Another optional element can be found in the sequence diagram of the second variant depicted in Figure 11. Thus, the recursive message call (called here *selfcall*) of the MP3Player is only alive if the second variant is selected. On account of this, the add operation (called here *add*) located in the MP3Player is only alive, if the second variant is selected.

However, those requirements can be expressed through the Ambiguity Concept as follows:

```
A3 = (project+ownedElement, xor,
{{MP3Playlist, connection, lifeline, call}, {selfcall, add}})
```

The difference between the ambiguity `A3` and `A2` or `A1` (see further down) is that the *choicevalues* are a set of elements instead of a single one. This ambiguity expresses exactly which elements can be selected and are needed at the same time related to the chosen variant. However, it is possible to express the two variants with the Ambiguity Concept as it has been shown in chapter 3.2.1 Relations between Ambiguities. In that approach more than one ambiguity must be defined and a reasoner is needed to detect valid combinations of ambiguous elements. This is due to the fact that the *choices* resulting from the defined ambiguities are combined, leading to an increased number of *choices*. However, in the list approach mentioned above, the ambiguity consists of only two *choices*: the first or the second list. Thus, the list approach is more generic and efficient due to a more restricted number of *choices*.

In theory, this issue is solved through the Ambiguity Concept (see `A3`) due to the fact that the concept of a *choicevalue* does not have a specific type. Thus, concerning the case study, a grouping mechanism depending on a variation point can reduce the number of *choices* due to fewer generated combinations of *choicevalues*.

In practice (respectively concerning the case study) this problem occurred due to a huge number of optional elements for the whole project. This means that if there are a lot of optional elements, such as sequence diagrams or classes, those selections depend on a particular product configuration. Ambiguity `A8` (5.2.1.2 Behavioural Changes, p. 76) of the case study describes that problem in a very simplified manner. In `A8` there exists only one *choicevalue*, respectively one optional sequence diagram. Concerning the whole case study project in the RSA there are a huge number of owned elements and optional elements. Thus, a lot of *choices* are generated due to all possible combinations of optional and mandatory elements. This leads to performance problems as described in 5.3.2 Problems.

Concerning the *Ambiguitymanager*, *choicevalue* elements must have a specific type. The list approach can be realised if the *Ambiguitymanager* offers the possibility to add lists of elements ignoring the difference in types. Hence, dealing with elements without knowing their specific structure can result in a greater effort in terms of individual implementations. This issue will also be discussed in 6.4 Complex Reverse Ambiguities.

6.3.2 Grouping Mechanism of Choices

Another limitation of the *Ambiguitymanager* is that as of yet no mechanism for elements, which must selected together (e.g. in product configuration of a product line), exists. Such a grouping mechanism for ambiguities, which depend on another, has yet to be created. An approach to add dependencies among certain *choices* of ambiguities (grouping mechanism of *choices*) is required.

In the case study, this issue was mentioned in the context of definitions of ambiguities in terms of *require* statements of features. In the *Ambiguitymanager* it is impossible to add such groups of *choices*. The output of the reasoning process, however, provides some indications which ambiguity *choices* can be selected together even if their ambiguities are not depending on the same model element and the same feature (location). In the feature model depicted in Figure 29 of the case study, one finds the feature *Other* of the variation point *Vehicle management*. However, *Other* enfolds four sub-features which partially require one another. The *Ambiguitymanager* does not offer a functionality to add a dependency between those features.

The previous subchapter describes a list approach for *choicevalues*. However, that section deals with the introduction of a grouping mechanism for *choices*. That means that an ambiguity *choice* can only be considered as a valid *choice* if it is selected with another *choice* of another independent ambiguity. In this case, *independent* means that their ambiguities do not belong to the same location. In Addition, *valid* is correct concerning from the perspective of a modelling aspect and not in terms of an UML rule. The latter can be solved via a connected reasoner. The following section illustrates this problem with the aid of a concrete example taken from the mp3 player scenario in 3.1.

The problem presented here is that the MP3Player has an optional association to the MP3Playlist, since there are two variants (Figure 8 or Figure 10) of the scenario. However, this association is only alive if the designer takes the first variant depicted in Figure 8. The UML feature for an association of a class is called *ownedAttribute*.

That ambiguity can be expressed in the following way:

```
A1 = (MP3Player+ownedAttribute, optional, {MP3Playlist})
```

Additionally, there are two *choices* resulting from that ambiguity. They are illustrated in Figure 33.

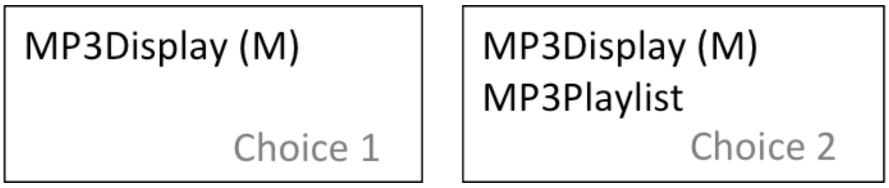


Figure 33: Choices of the first ambiguity.

In addition, there are two options where to store the add operation. In the first variant it is located in the MP3Playlist, in the second variant one finds it in the MP3Player. Thus, the following ambiguity can be derived as follows:

```
A2 = (add()+class, xor, {MP3Player, MP3Playlist})
```

Additionally, there are two *choices* resulting from the second ambiguity. They are illustrated in Figure 34.

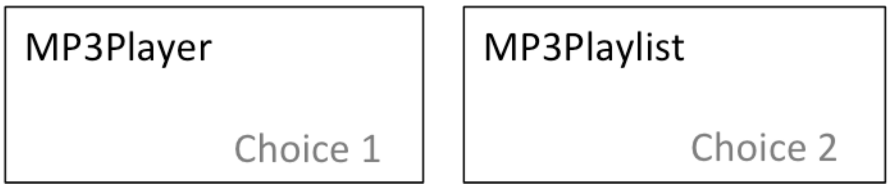


Figure 34: Choices of the second ambiguity.

There are only two valid combinations of *choices* for the model to express the first or the second variant of the scenario.

The selection of Choice 1 of the first ambiguity and the Choice 1 of the second can be considered as valid. This combination expresses that there is only one association from the MP3Player to the MP3Display and the add operation is located in the MP3Player. This realises the second variant of the mp3 player scenario (with only two components) exactly.

The selection of Choice 2 of the first ambiguity and the Choice 2 of the second can be considered valid as well. This combination expresses that there is an association from the MP3Player to the MP3Playlist and the add operation is located in the MP3Playlist. This realises the first variant of the mp3 player scenario (with three components) exactly.

However, due to the *choice* generation introduced in 3.2.1 Relations between Ambiguities, all combinations are valid, since the association and the location of the add operation do not affect one another. In the RSA the MP3Playlist is always alive, so selecting that class as the owner for the add operation remains a valid UML rule to select. If this was not the case, the reasoner would detect the invalid *choice* for the ownership because of the missing *choicevalue* MP3Playlist.

6.3.3 Choice in Multiple Ambiguities

A UML element can be referenced by different *choicevalues* of ambiguities. There are situations in which those ambiguities are independent from one another concerning any relations in a model. There are no implicit dependencies between them. In addition, they do not have any points of contract in the reasoning process.

The output of a reasoning process gives information about invalid combinations of *choices* if they are related to one another depending on UML constraints (rule). However, the output does not detect invalid combinations of *choices* if their depending ambiguities have no explicit or implicit dependencies. In the *Ambiguitymanager choicevalues* are unique elements for a location. Thus, there can be *n choicevalues* for different locations, which reference to the same element (implicit dependency). Nevertheless, if *choices* contain same elements in that way, they actually depend on one another. This is due to the fact, that *choices* for a location can affect other locations with the same referenced UML element. That dependency is ignored during the definition of ambiguities and in any conducted reasoning process.

However, the reasoning process can be improved to find such dependencies. Additionally, the architecture of the *Ambiguitymanager*, respectively the model package, can be modified to treat that *choicevalues* as unique elements in the whole program.

6.3.4 Solution with Constraints

In the Ambiguity Concept ambiguities are considered as independent elements. They do not have any relations to one another. A reasoner checks the consistency of combinations, respectively of their *choices*. However, the output of the reasoning process is a collection of valid and invalid combinations of all ambiguity *choices*. If *choices* of different ambiguities are related to each other, their *choicevalues*, respectively the ambiguous model elements in the depending modelling project, are related as well. An example for such a relation is provided through a condition/constraint in Table 1 and is described in 2.2.1 Consistency Rules. Those rules describe dependencies of UML elements to prove a model as valid.

However, concerning the case study and the product line approach a decision for a feature is made due to a requirement in a product configuration. To deem a model as valid concerning a requirement, it can also be solved through conditions for ambiguities. Those conditions can be defined as constraints for ambiguity *choices*. This means one has to decide, which ambiguity *choices* can be selected together to fulfil a product line feature. Hence, the dependency between ambiguities can be expressed through those constraints.

6.4 Complex Reverse Ambiguities

This section is concerned with complex reverse ambiguities. A reverse ambiguity is generated by a definition of an ambiguity.

The *Ambiguitymanager* und the initial Ambiguity Concept can only handle single *choicevalues*. As described in the previous sections, the Ambiguity Concept can be extended to offer such a grouping functionality.

However, the *Ambiguitymanager* is restricted, since *choicevalue* elements must have a specific type. The data structure of the *Ambiguitymanager* can be modified to allow lists of *choicevalues*. Hence, a specific type or even a meta-type can be advantageous. Since such a group of *choicevalue* elements can be defined, a reverse ambiguity for each element must be generated.

A reverse ambiguity's location consists of the *choicevalue* and the opposite feature of the original feature. Without any type restrictions, it may be the case that no opposite feature for one of the *choicevalues* exists, due to the fact, that there is e.g. no definition in the UML specification. Hence, dealing with elements without knowing their specific structure or type may result in additional effort in terms of individual implementations or even uncontrollable side effects.

7 Summary and Conclusion

A software designer has the ability to define a system using the UML. However, a concept to document design decisions along the modelling process or to try different decisions is yet to be developed. Furthermore, such a concept could be used to check the consistency of unsolved decisions and to detect valid or invalid combinations.

This thesis introduced the Ambiguity Concept. This concept can be used to add different design decision, so called ambiguities, to design models. A consistency checker (reasoner) is used to detect inconsistencies in a model with ambiguities.

The Unified Modeling Language (UML) is the de facto standard modelling language to design a system in terms of its behavioural and structural aspects. Due to this, the Ambiguity Concept is discussed in terms of the UML.

In addition, the implementation of the *Ambiguitymanager*, a plugin for the IBM Rational Software Architect, was rpresented. The *Ambiguitymanager* can be used to add ambiguities to UML models and to conduct reasoning steps.

Additionally, a huge case study employing the *Ambiguitymanager* was implemented. This way, strengths and weaknesses of the Ambiguity Concept and the *Ambiguitymanager* were discussed.

7.1 Conclusion

The Ambiguity Concept introduced in this thesis offers strengths and weaknesses concerning expressions of unsolved decisions to design models. It has the advantage that it offers the possibility to define optional elements instead of re-modelling whole entire parts depending on a decision. With the *Ambiguitymanager* one can select particular model elements of a complex model and add certain ambiguities instead of re-modelling the entire diagram.

Another advantage concerning software product line approaches, is that resulting *choices* for a location are generated automatically and do not have to be configured by a designer as it would be the case in a product configuration process.

Although all examples are realised with the UML, the Ambiguity Concept is applicable for any kind of modelling language as long as the language is based on a well defined meta-model.

However, the Ambiguity Concept is even capable of dealing with grouped elements as was proven in 6.3.1 Grouping Mechanism Choicevalues. Concerning the case study, a weakness of the *Ambiguitymanager* is that it does not offer a mechanism as it is required in depending features in a variation point. A grouping mechanism for ambiguities or model elements, which require one another, is missing. Additionally, it is required to add any kind of dependencies to ambiguities, *choices*, and *choicevalues* to grant a designer additional flexibility.

The *Ambiguitymanager* does not offer any visualisation in the graphical editor of defined ambiguities and their possible effect on the whole model. This problem was encountered during the implementation of the case study when huge sequence diagrams were modelled. There is a large number of newly created lifelines, calls between them, guard conditions, etc. and thus, it would be very hard to gain an overview about a model, respectively a sequence diagram, without seeing defined components and their relations. Due to this, the *Ambiguitymanager* does not yet offer any benefit in terms of re-modelling lesser aspects in terms of completely new defined complex sequence diagrams.

In conclusion, the implementation of a larger case study shows aspects of the Ambiguity Concept, which did not emerge in smaller modelling projects. Some of them are results of the reasoning process and the huge number of generated *choices*. There exists a scalability problem, as mentioned in 5.3.2 Problems, as a result of the huge number of generated *choices* and its visualisation via the RSA.

The implementation of the case study proved that the Ambiguity Concept enables users to express complex aspects of variability in design models and to handle a wide range of design decisions. In addition, the *Ambiguitymanager* can be extended in terms of new definitions of model elements and their features. Unfortunately, the UML specification is very complex. There is a huge number of definitions, so implementing all certain cases for any UML would take much effort and a tremendous amount of knowledge. Due to this, a generic approach in terms of the extraction of feature definitions is not realizable in a simple way.

Although the *Ambiguitymanager* has its weaknesses, they can be solved by a functionality to add more dependencies to ambiguities, *choices*, and *choicevalues*. Additionally, the usability of the *Ambiguitymanager can* be improved by a grouping mechanisms and extended graphical visualisation.

7.2 Future Work

The implementation of the case study clarifies strengths but also weaknesses of the Ambiguity Concept. As shown in the previous section, the Ambiguity Concept can handle *choicevalues*, which contain a group of elements. Further work is required in order to extend the *Ambiguitymanager* by introducing a grouping mechanism of depended ambiguities and model elements. Additionally, it is essential to add more dependencies to ambiguities, *choices*, and *choicevalues*.

Furthermore, a graphical visualisation of effects on the whole model for the RSA is inevitable. By doing this, the designer can gain an overview about his ambiguities and design decisions immediately.

The *determinations* resulting from a reasoning process can encapsulate additional knowledge which can be derived. This means concretely, that if there is no valid *determination* for a given constraint (meaning there is no *choice* left that fulfils defined requirements) there must be a general design fault in the model. Hence, the model can be considered as invalid by its definition.

Additionally, it can be helpful to provide indications of values or elements, which are defined initially as optional, but become mandatory by occurring in any valid *choice*. However, if a value or element never occurs in a valid *choice*, it can be an indication of an invalid value, thus, a modelling fault.

Determinations can be used to identify values or elements, which are defined as mandatory or required, but exclude other required values, resulting in a lack of *choice* comprising all required ones.

The *determinations* offer some additional important and useful knowledge that should be made available to a designer.

Figures

Code Examples

Tables

References

[1] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Inc. 2005.

[2] H. Gomaa, *Designing Software Product Lines with UML.,* Software Engineering Workshop – Tutorial Notes, 29th Annual IEEE/NASA pp. 160-216, 2005.

[3] A. Egyed, D. S. While, *Support for Managing Design-time Decisions*, IEEE Transactions on Software Engineering, vol. 32, no. 5, pp. 299-314, May 2006.

[4] G. Booch, J. Rumbaugh, I. Jacobson, *Das UML Benutzerhandbuch*, Addison-Wesley Verlag, 2006.

[5] A. Egyed, *Instant Consistency Checking for the UML*, Proceedings 28th International Conference on Software Engineering (ICSE), May 2005.

[6] A. Egyed, N. Medvidovic, *Consistent Architectural Refinement and Evolution using the Unified Modeling Language*, Proceedings of the first Workshop on Describing Software Architecture with UML, co-located with ICSE 2001, pp. 83-87, Toronto, Canada, May 2001.

[7] A. Egyed, *Scalable Consistency Checking between Diagrams - The ViewIntegra Approach*, Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE), pp. 387-390, San Diego, USA, September 2001.

[8] A. Egyed, *Taming Ambiguity to Overcome the Model Consistency Barrier*, European Conference on Software Engineering and Foundations of Software Engineering (ESEC/FSE), Vienna, Austria, September 2001.

[9] A. Egyed, *UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models*, Proceedings 29th International Conference on Software Engineering (ICSE), May 2007.

[10] D. Benavides, S. Segura, A. Ruiz Cortés, *Automated analysis of feature models 20 years later: A literature review*, Inf. Syst. vol. 35, no. 6, pp. 615-636, 2010.

[11] D. Le Berre, SAT4J solver, http://www.sat4j.org, accessed July 2012.

[12] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1995.

[13] R. Barták, *Constraint Programming: In Pursuit of the Holy Grail*, Proceedings of the Week of Doctoral Students (WDS'99), Part IV (Invited Lecture), pp. 555-564, Prague, June 1999.

[14] P. Henteryck, *Strategic Directions in Constraint Programming*,

ACM Computing Surveys, vol. 28, no. 4, 1996.

[15] S. Easterbrook, M. Chechik, *A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints*, Proceedings 23rd International Conference on Software Engineering, pp. 411-420, May 2001.

[16] IBM, *Rational Software Architect*, http://www.ibm.com/developerworks/rational/products/rsa/2012, accessed July 2012.

[17] IBM, *Rational Software*, http://www-01.ibm.com/software/de/rational/, accessed July 2012.

[18] The Eclipse Foundation, *Eclipse Modeling Framework Project*, http://www.eclipse.org/modeling/emf/2012, accessed July 2012.

[19] The Eclipse Foundation, *Eclipse Modeling Project*, http://www.eclipse.org/modeling/2012, accessed July 2012.

[20] T. Ziadi, J. - M. Jezequel, *Software product line engineering with the uml: Deriving products.,* Software Product Lines, 2006, pp. 557–588.

[21] L. M. Northrop, *Sei's software product line tenets.,* IEEE Software vol. 19, no. 4, 2002, pp. 32–40.

[22] P. Clements, L. M. Northrop, *Software Product Lines: Practices and Patterns, The SEI Series in Software Engineering*, Addison-Wesley, Boston, 2002.

[23] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach.*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[24] A. Capozucca, B. H. C. Cheng, N. Guelfi, P. Istoan, *OO-SPL modelling on the focused case study*, September 2011.

[25] K. Pohl, G. Böckle, F. Van Der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer-Verlag Berlin Heidelberg, Germany, 2005.

[26] Software product line conference – hall of fame. http://splc.net/fame.html, accessed July 2012.

[27] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wąsowski, *Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches*, Proceedings 6th International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, 2012, pp. 173-182.

[28] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, *Feature-oriented domain analysis (FODA) feasibility study.*, Technical report, CMU/SEI-90TR-21, 1990.

[29] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. Reuse-Driven Software Processes Guidebook, Version 02.00.03, 1993.

[30] M. Svahnberg, J. Bosch, *Issues Concerning Variability in Software Product Lines*, Proceedings of the Third International Workshop on Software Architectures for Product Families, pp. 50-60, 2000.

[31] C. Krueger, *The BigLever Software Gears Unified Software Product Line Engineering Framework.*, Proceedings of the International Software Product Line Conference (SPLC), pp. 353–353, 2008.

[32] D. Beuche, H. Papajewski, W. Schröder-Preikschat, *Variability Management with Feature Models.*, *Science of Computer Programming (SCP)*, vol. 53, no. 3, pp. 333–352, 2004.

[33] S. Apel, C. Kästner, *An Overview of Feature-Oriented Software Development.*, Journal of Object Technology vol. 8, no. 5, pp. 49-84, 2009.

[34] IBM, "Eclipse Project", http://www.eclipse.org/org/, 2001.

[35] The Eclipse Foundation, Eclipse Platform Release 3.7

http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fui%2Fpart%2FViewPart.html, accessed July 2012.

[36] org.eclipse.uml2.uml.UMLPackage.Literals http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.uml2.doc%2Freferences%2Fjavadoc%2Forg%2Feclipse%2Fuml2%2Fuml%2FUMLPackage.Literals.html, accessed July 2012.

[37] A. Capozucca, B. H. C. Cheng, N. Guelfi, P. Istoan, G. Mussbacher, *Requirements definition document of focused case study*, June 2011.

http://cserg0.site.uottawa.ca/cma2011/CaseStudy.pdf accessed July 2012.

Curriculum Vitae

Name:	Franziska Oellerer
Academic degree:	Bachelor of Science (BSc.)
Date of birth:	1985/10/13
Place of birth:	Hanover, Germany
Place of residence:	Viktoriastr. 5A, 30451 Hanover, Germany
Email address:	info@franzolina.de
Languages	German (native language), English (fluent), French (conversational), Swedish (basic knowledge)
Professional Experience	
	January 2006–Present VIVA Models, Berlin: Mannequin
	May 2008–January 2010 University of Hildesheim, Software Systems Engineering: Student assistant
	October 2006–October 2010 Online journal langeleine media: Uncomplimentary Freelancer
	August 2009–October 2009 itemis AG Lünen: Internship

July 2009

iProCon GmbH/University of Hildesheim: Docent for Java (Freelancer)

September 2007–October 2007

Countryhotel Reguengo Portugal:

Temporary Employment

April 2005–July 2005

Stadtkindverlag stadtkind Magazine:

Internship

December 2004–April 2005

Ralf Mohr Photographie: Personal assistant for photography and graphic design

Education and Training

September 2010–Present

Johannes Kepler University of Linz: Student in Computer Science/Software Engineering, Master of Science/Diplom Ingenieurin

October 2007–August 2010

University of Hildesheim: Student in Information Management/Information Technology, Bachelor of Science

Winter Term 2006, VHS Hannover,

Course: *Science of Journalism 1*

Summer Term 2004, VHS Hannover,

Course: *Image-Editing with Photoshop*

Sworn Declaration

I hereby declare under oath that the submitted Master's thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, January 17th, 2013

Franziska Öllerer B. Sc.